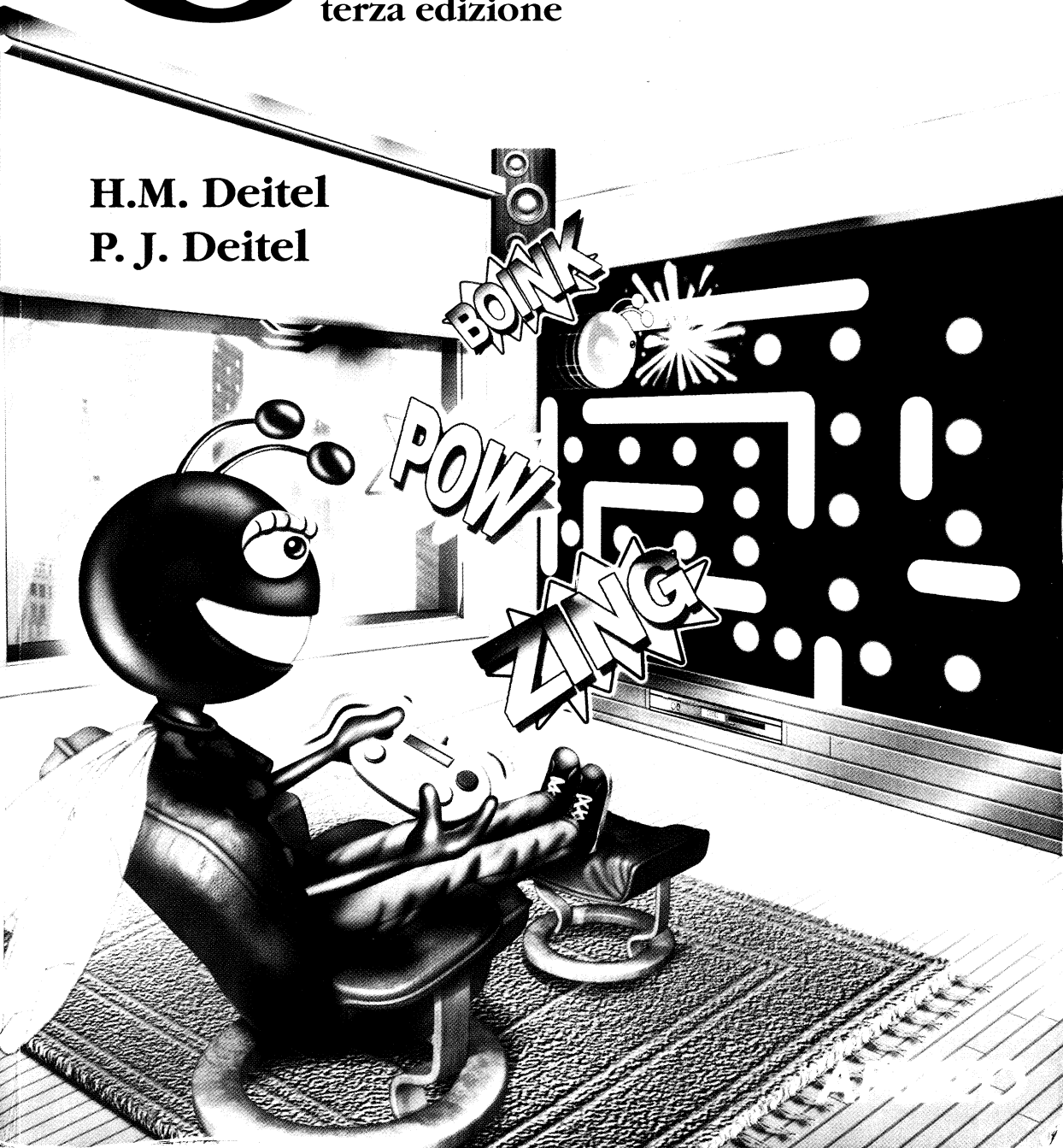
APOGEO education

C Corso completo di programmazione

terza edizione

H.M. Deitel
P. J. Deitel



VISIT…

C
Corso completo di programmazione, terza edizione

Titolo originale:
C How to Program, Fifth Edition

Autori:
Harvey M. Deitel, Paul J. Deitel

Published by arrangement with the original publisher, **PRENTICE HALL, INC.**, a Pearson Education Company
Copyright © **2007 – PRENTICE HALL, INC.**

Copyright per l'edizione italiana © **2007 – APOGEO s.r.l.**
Socio Unico Giangiacomo Feltrinelli Editore s.r.l.
Via Natale Battaglia, 12 – 20127 Milano (Italy)
Telefono: 02-289981 – Telefax: 02-26116334
Email **education@apogeonline.com**
U.R.L. **http://www.apogeonline.com**

ISBN-13 978-88-503-2633-4

Traduzione e revisione: Ivan Scagnetto
Impaginazione elettronica: Grafica editoriale – Vimercate
Editor: Alberto Kratter Thaler
Copertina e progetto grafico: Enrico Marcandalli
Responsabile di produzione: Vitiano Zaini

Tutti i diritti sono riservati a norma di legge e a norma delle convenzioni internazionali. Nessuna parte di questo libro può essere riprodotta con sistemi elettronici, meccanici o altri, senza l'autorizzazione scritta dell'Editore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, comma 4, della legge 22 aprile 1941 n. 633 ovvero dell'accordo stipulato tra SIAE, AIE, SNS e CNA, CONFARTIGIANATO, CASA, CLAAI, CONFCOMMERCIO, CONFESERCENTI il 18 dicembre 2000.

Le riproduzioni a uso differente da quello personale potranno avvenire, per un numero di pagine non superiore al 15% del presente volume, solo a seguito di specifica autorizzazione rilasciata da AIDRO, C.so di Porta Romana, n. 108, – 20122 Milano, telefono 02 89280804, telefax 02 892864, e-mail aidro@iol.

Finito di stampare nel mese di giugno 2007
da Legoprint – Lavis, Trento

Sommario

PREFAZIONE XI
Lo scopo di questo libro xi
La metodologia di insegnamento xi
Panoramica sul libro xiv
Ringraziamenti xix
Gli autori xxii

CAPITOLO 1: INTRODUZIONE AI COMPUTER, A INTERNET E AL WEB 1
1.1 Introduzione 1
1.2 Che cosa è un computer? 3
1.3 L'organizzazione del computer 3
1.4 L'evoluzione dei sistemi operativi 4
1.5 I personal computer, i sistemi distribuiti e i sistemi client/server 5
1.6 I linguaggi macchina, assembly e di alto livello 6
1.7 FORTRAN, COBOL, Pascal e Ada 7
1.8 La storia del C 8
1.9 La libreria standard del C 9
1.10 Il C++ 10
1.11 Java 10
1.12 BASIC, Visual Basic, Visual C++, C# e .NET 11
1.13 La chiave dell'evoluzione del software; la tecnologia a oggetti 12
1.14 Le basi di un tipico ambiente di sviluppo di un programma C 13
1.15 L'evoluzione dell'hardware 16
1.16 La storia di Internet 16
1.17 La storia del World Wide Web 18
1.18 Note generali sul C e su questo libro 18
Esercizi di autovalutazione 19
Risposte agli esercizi di autovalutazione 20
Esercizi 20

CAPITOLO 2: INTRODUZIONE ALLA PROGRAMMAZIONE IN C 23
2.1 Introduzione 23
2.2 Un semplice programma C: visualizzare una riga di testo 23
2.3 Un altro semplice programma C: sommare due interi 28
2.4 Nozioni sulla memoria 33
2.5 L'aritmetica del C 34
2.6 Prendere delle decisioni: gli operatori di uguaglianza e relazionali 37
Esercizi di autovalutazione 42
Risposte agli esercizi di autovalutazione 44
Esercizi 45

CAPITOLO 3: LO SVILUPPO DI PROGRAMMI STRUTTURATI IN C 49
3.1 Introduzione 49
3.2 Gli algoritmi 49
3.3 Lo pseudocodice 50
3.4 Le strutture di controllo 51
3.5 Il comando di selezione if 53
3.6 Il comando di selezione if...else 54
3.7 Il comando di iterazione while 58
3.8 Formulazione degli algoritmi: studio di un caso 1 (iterazione controllata da un contatore) 60
3.9 Formulazione degli algoritmi con processo top-down per raffinamenti successivi: studio di un caso 2 (iterazione controllata da un valore sentinella) 62
3.10 Formulazione degli algoritmi con processo top-down per raffinamenti successivi: studio di un caso 3 (strutture di controllo nidificate) 69
3.11 Gli operatori di assegnamento 74
3.12 Gli operatori di incremento e di decremento 75
Esercizi di autovalutazione 78
Risposte agli esercizi di autovalutazione 79
Esercizi 81

CAPITOLO 4: IL CONTROLLO DEL PROGRAMMA IN C 91
4.1 Introduzione 91
4.2 Gli elementi dell'iterazione 91
4.3 Iterazione controllata da un contatore 92
4.4 Il comando di iterazione for 94
4.5 Il comando for: note e osservazioni 97
4.6 Esempi di utilizzo del comando for 98
4.7 Il comando di selezione multipla switch 102
4.8 Il comando di iterazione do...while 108
4.9 Le istruzioni break e continue 110
4.10 Gli operatori logici 112
4.11 Confondere gli operatori di uguaglianza (==) e di assegnamento (=) 115
4.12 Riassunto della programmazione strutturata 116
Esercizi di autovalutazione 121
Risposte agli esercizi di autovalutazione 123
Esercizi 124

CAPITOLO 5: LE FUNZIONI IN C 131
5.1 Introduzione 131
5.2 I moduli di programma in C 131
5.3 Le funzioni della libreria matematica 133
5.4 Le funzioni 133
5.5 Le definizioni di funzione 135
5.6 I prototipi di funzione 139
5.7 I file di intestazione 142

5.8 I file di intestazione 144
5.9 Invocare le funzioni: chiamata per valore e per riferimento 144
5.10 Generazione di numeri casuali 145
5.11 Esempio: un gioco d'azzardo 151
5.12 Le classi di memoria 154
5.13 Le regole di visibilità 157
5.14 La ricorsione 161
5.15 Esempio di utilizzo della ricorsione: la serie di Fibonacci 165
5.16 Ricorsione e iterazione 168
Esercizi di autovalutazione 171
Risposte agli esercizi di autovalutazione 173
Esercizi 175

CAPITOLO 6: I VETTORI IN C 185
6.1 Introduzione 185
6.2 I vettori 185
6.3 La dichiarazione dei vettori 187
6.4 Esempi sui vettori 187
6.5 Passare i vettori alle funzioni 202
6.6 L'ordinamento dei vettori 208
6.7 Studio di un caso: calcolare la media, la mediana e la moda usando i vettori 210
6.8 La ricerca nei vettori 216
6.9 I vettori multidimensionali 222
Esercizi di autovalutazione 230
Risposte agli esercizi di autovalutazione 231
Esercizi 232
Esercizi sulla ricorsione 243

CAPITOLO 7: I PUNTATORI IN C 245
7.1 Introduzione 245
7.2 Dichiarazione e inizializzazione dei puntatori 245
7.3 Gli operatori sui puntatori 246
7.4 La chiamata per riferimento delle funzioni 248
7.5 Utilizzare il qualificatore const con i puntatori 253
7.6 L'ordinamento a bolle utilizzando una chiamata per riferimento 261
7.7 L'operatore sizeof 264
7.8 Le espressioni con i puntatori e l'aritmetica dei puntatori 267
7.9 La relazione tra i puntatori e i vettori 270
7.10 I vettori di puntatori 274
7.11 Studio di un caso: simulazione di un mescolatore e distributore di carte 275
7.12 I puntatori a funzioni 281
Esercizi di autovalutazione 287
Risposte agli esercizi di autovalutazione 288
Esercizi 289
Sezione speciale: costruite il vostro computer 293

CAPITOLO 8: I CARATTERI E LE STRINGHE IN C 305
8.1 Introduzione 305
8.2 I concetti fondamentali delle stringhe e dei caratteri 305
8.3 La libreria per la gestione dei caratteri 307
8.4 Le funzioni per la conversione delle stringhe 313
8.5 Le funzioni della libreria per l'input/output standard 318
8.6 Le funzioni per la manipolazione delle stringhe incluse nella libreria per la gestione delle stringhe 323
8.7 Le funzioni di confronto incluse nella libreria per la gestione delle stringhe 326
8.8 Le funzioni di ricerca incluse nella libreria per la gestione delle stringhe 328
8.9 Le funzioni per la manipolazione della memoria incluse nella libreria per la gestione delle stringhe 335
8.10 Le altre funzioni della libreria per la gestione delle stringhe 340
Esercizi di autovalutazione 341
Risposte agli esercizi di autovalutazione 342
Esercizi 343
Sezione speciale: esercizi di manipolazione avanzata delle stringhe 346
Un progetto impegnativo per la manipolazione delle stringhe 350

CAPITOLO 9: LA FORMATTAZIONE DELL'INPUT/OUTPUT IN C 351
9.1 Introduzione 351
9.2 Gli stream 351
9.3 Formattare l'output con printf 352
9.4 Visualizzare gli interi 352
9.5 Visualizzare i numeri in virgola mobile 354
9.6 Visualizzare le stringhe e i caratteri 356
9.7 Gli altri indicatori di conversione 357
9.8 Visualizzare con le dimensioni di campo e le precisioni 359
9.9 Utilizzare i flag nella stringa di controllo del formato della printf 362
9.10 Visualizzare i letterali e le sequenze di escape 365
9.11 Formattare l'input con scanf 366
Esercizi di autovalutazione 373
Risposte agli esercizi di autovalutazione 374
Esercizi 375

CAPITOLO 10: LE STRUTTURE, LE UNIONI, LA GESTIONE DEI BIT E LE ENUMERAZIONI IN C 379
10.1 Introduzione 379
10.2 La definizione delle strutture 379
10.3 Inizializzare le strutture 382
10.4 Accedere ai membri delle strutture 382
10.5 Usare le strutture con le funzioni 384
10.6 typedef 385
10.7 Esempio: simulazione di un mescolatore e distributore di carte ad alta efficienza 386

10.8 Le unioni 389
10.9 Gli operatori bitwise 391
10.10 I campi di bit 401
10.11 Le costanti di enumerazione 405
Esercizi di autovalutazione 406
Risposte agli esercizi di autovalutazione 408
Esercizi 409

CAPITOLO 11: L'ELABORAZIONE DEI FILE IN C 413
11.1 Introduzione 413
11.2 La gerarchia dei dati 413
11.3 I file e gli stream 415
11.4 Creare un file ad accesso sequenziale 416
11.5 Leggere i dati da un file ad accesso sequenziale 422
11.6 I file ad accesso casuale 427
11.7 Creare un file ad accesso casuale 428
11.8 Scrivere i dati in modo casuale in un file ad accesso casuale 430
11.9 Leggere i dati in modo casuale da un file ad accesso casuale 433
11.10 Studio di un caso: un programma per l'elaborazione delle transazioni 435
Esercizi di autovalutazione 442
Risposte agli esercizi di autovalutazione 443
Esercizi 444

CAPITOLO 12: LE STRUTTURE DI DATI IN C 449
12.1 Introduzione 449
12.2 Le strutture ricorsive 450
12.3 Allocazione dinamica della memoria 451
12.4 Le liste concatenate 452
12.5 Le pile 461
12.6 Le code 467
12.7 Gli alberi 474
Esercizi di autovalutazione 480
Risposte agli esercizi di autovalutazione 481
Esercizi 482

CAPITOLO 13: IL PREPROCESSORE DEL C 505
13.1 Introduzione 505
13.2 La direttiva del preprocessore #include 505
13.3 La direttiva del preprocessore #define: le costanti simboliche 506
13.4 La direttiva del preprocessore #define: le macro 506
13.5 La compilazione condizionale 508
13.6 Le direttive del preprocessore #error e #pragma 510
13.7 Gli operatori # e ## 510
13.8 I numeri di riga 510
13.9 Le costanti simboliche predefinite 511
13.10 Le asserzioni 511

Esercizi di autovalutazione 512
Risposte agli esercizi di autovalutazione 513
Esercizi 513

CAPITOLO 14: ARGOMENTI AVANZATI 515
14.1 Introduzione 515
14.2 Redirezionare l'input/output su sistemi UNIX e Windows 515
14.3 Gli elenchi variabili di argomenti 516
14.4 Usare gli argomenti della riga di comando 519
14.5 Note sulla compilazione di programmi formati da vari file sorgente 520
14.6 Chiusura dei programmi con exit e atexit 522
14.7 Il qualificatore di tipo volatile 524
14.8 I suffissi per le costanti intere e in virgola mobile 524
14.9 Ancora sui file 525
14.10 La gestione dei segnali 527
14.11 Allocazione dinamica della memoria: le funzioni calloc e realloc 530
14.12 Il salto incondizionato con goto 531
Esercizi di autovalutazione 532
Risposte agli esercizi di autovalutazione 533
Esercizi 533

CAPITOLO 15: INTRODUZIONE AL C99 535
15.1 Introduzione 535
15.2 Supporto del C99 536
15.3 I nuovi file header del C99 537
15.4 Commenti introdotti da // 538
15.5 Intercalare dichiarazioni e codice eseguibile 538
15.6 Dichiarare una variabile nell'intestazione di un comando for 539
15.7 Inizializzatori designati e letterali composti 541
15.8 Il tipo di dati bool 544
15.9 Tipo int implicito nelle dichiarazioni di funzioni 546
15.10 Numeri complessi 547
15.11 Vettori di lunghezza variabile 549
15.12 Altre caratteristiche del C99 550
15.13 Risorse disponibili su Internet e sul Web 554
Esercizi di autovalutazione 555
Risposte agli esercizi di autovalutazione 556
Esercizi 556

APPENDICE A: RISORSE SU INTERNET E NEL WORLD WIDE WEB 557

APPENDICE B: RISORSE SUL C99 DISPONIBILI SU INTERNET E NEL WORLD WIDE WEB .. 559

APPENDICE C: TABELLA DI PRIORITÀ DEGLI OPERATORI 563

APPENDICE D: L'INSIEME DEI CARATTERI ASCII 565

APPENDICE E: I SISTEMI NUMERICI 567

APPENDICE F: RISORSE SULLA LIBRERIA STANDARD DEL C 579

APPENDICE G: LA LIBRERIA STANDARD 581

APPENDICE H: PROGRAMMARE GIOCHI: RISOLVERE IL SUDOKU 619

INDICE ANALITICO 629

Prefazione

Benvenuti nel mondo dello Standard ANSI/ISO C. Alla Deitel & Associates scriviamo libri di testo di livello universitario sui linguaggi di programmazione e libri per i professionisti e lavoriamo duramente per mantenerli aggiornati con un flusso costante di nuove edizioni. Scrivere la quarta edizione di questo libro è stato un piacere: esso contiene tutto ciò di cui sia gli insegnanti che gli studenti possano aver bisogno per sperimentare un'esperienza educativa ricca di contenuti, interessante, stimolante e anche divertente. Abbiamo affinato lo stile della scrittura, gli aspetti pedagogici e il modo di scrivere il codice. Inoltre, abbiamo incluso una panoramica sul libro in questa prefazione, in modo da aiutare gli insegnanti, gli studenti e i professionisti a rendersi conto della ricchezza dei contenuti sul linguaggio C proposti da questo libro. In questa prefazione presentiamo le convenzioni che utilizzeremo nel corso del testo.

Lo scopo di questo libro

Il Professor Harvey M. Deitel ha tenuto corsi universitari introduttivi sulla programmazione per 20 anni, puntando soprattutto a insegnare come sia possibile sviluppare programmi ben scritti e ben strutturati. Molti dei suoi insegnamenti riguardano i concetti fondamentali della programmazione, con particolare enfasi sull'utilizzo efficace dei comandi di controllo e sulla funzionalità. Questi argomenti vengono presentati all'interno di questo libro esattamente nel modo in cui il Professor Deitel li ha sempre proposti ai propri studenti universitari.

Il nostro obiettivo era chiaro: offrire un testo sul C destinato ai corsi universitari a livello introduttivo, per quegli studenti che non hanno alcuna esperienza nell'ambito della programmazione, pur fornendo, allo stesso tempo, la completezza teorica e pratica richiesta ai tradizionali corsi avanzati di C.

Il testo segue lo standard ANSI C; tenete presente che molte funzionalità dell'ANSI C non sono implementate nelle versioni pre-ANSI del C. Per avere informazioni più dettagliate sul linguaggio vi conviene consultare il manuale di riferimento del vostro sistema o procurarvi una copia del documento ANSI/ISO 9899: 1990, "American National Standard for Information Systems-Programming Language C", dell'American National Standards Institute, 11 West 42nd Street, New York, New York 10036.

La metodologia di insegnamento

Questo libro contiene un'ampia varietà di esempi, esercizi e progetti che prendono spunto da molte situazioni e che offrono agli studenti la possibilità di risolvere problemi reali. Il libro prende in esame i principi della progettazione del software, insistendo sull'importanza della chiarezza dei programmi ed evitando l'uso di terminologia complessa a favore di esempi chiari e diretti, il cui codice sia stato collaudato sulle piattaforme C più diffuse.

L'apprendimento attraverso il codice

Il libro presenta una grande quantità di esempi basati sul codice. Ogni argomento viene presentato nell'ambito di un programma C completo e funzionante, seguito sempre da una o più finestre che mostrano l'output del programma in questione. Viene quindi usato il linguaggio per insegnare il linguaggio, e la lettura di questi programmi offre un'esperienza molto simile alla loro esecuzione reale su di un computer.

L'accesso al World Wide Web

Tutto il codice presente nel libro si trova anche in Internet, nel booksite abbinato a questo libro, all'indirizzo `http://www.apogeonline.com/libri/88-503-2633-5/scheda`. Il nostro consiglio è quello di scaricare tutto il codice, eseguendo poi ogni singolo programma via via che appare nel testo. Potete anche modificare il codice degli esempi e vedere cosa succede, imparando così a programmare programmando. Tutto questo materiale è protetto da diritti d'autore, quindi utilizzatelo liberamente per studiare il C, ma non pubblicatene alcuna parte senza l'esplicito permesso da parte degli autori e della casa editrice.

Obiettivi

Ogni capitolo inizia con la presentazione degli *Obiettivi*. Gli studenti possono così sapere in anticipo ciò che andranno ad apprendere e, alla fine della lettura del capitolo, potranno verificare se hanno raggiunto o meno questi obiettivi.

Il codice e gli esempi

Le funzionalità del C vengono presentate nell'ambito di programmi completi e funzionanti. Ogni programma è seguito dalle immagini degli output che vengono prodotti, così che gli studenti possano assicurarsi della correttezza dei risultati. I programmi presentati vanno da poche linee di codice a esempi composti da varie centinaia di righe. Gli studenti dovrebbero scaricare tutto il codice dal sito Web `http://www.apogeonline.com/libri/88-503-2633-5/scheda`, eseguendo poi ogni programma via via che questo viene presentato all'interno del testo.

Figure e immagini

Il libro offre un'ampia varietà di grafici, immagini e output di programmi. Nelle sezioni dedicate alle strutture di controllo, per esempio, appaiono vari diagrammi di flusso molto utili. [*Nota*: i diagrammi di flusso non vengono presentati come uno strumento di sviluppo, ma ricorriamo a una breve presentazione basata su questi diagrammi per specificare le singole operazioni di ogni struttura di controllo C.]

Consigli e suggerimenti

Il libro offre molti suggerimenti riguardanti la programmazione, per aiutare gli studenti a concentrarsi sugli aspetti più importanti dello sviluppo di programmi. Questi consigli vengono forniti attraverso le sezioni chiamate *Buona abitudine*, *Errore tipico*, *Collaudo e messa a punto*, *Obiettivo efficienza*, *Obiettivo portabilità*, *Ingegneria del software*.

Buona abitudine

Il concetto più importante per chi inizia a programmare è la chiarezza, e all'interno delle sezioni *Buona abitudine* vengono presentate delle tecniche per la scrittura di programmi chiari, comprensibili e più facilmente gestibili.

Errore tipico

Tutti gli studenti che affrontano per la prima volta un nuovo linguaggio tendono a commettere frequentemente gli stessi errori. Le sezioni *Errore tipico* aiutano gli studenti a evitare di commettere i più comuni.

Collaudo e messa a punto

Queste sezioni forniscono consigli circa le attività di test e debugging dei programmi C, anche a livello preventivo.

Obiettivo efficienza

In base alla nostra esperienza, insegnare agli studenti come scrivere programmi chiari e comprensibili deve costituire l'obiettivo più importante di qualsiasi corso di programmazione. Gli studenti, però, vogliono normalmente imparare a scrivere programmi che vengano eseguiti in modo veloce, che utilizzino poca memoria, che richiedano una minima quantità di comandi e che offrano prestazioni eccellenti. Le sezioni *Obiettivo efficienza* offrono suggerimenti su come migliorare le prestazioni dei propri programmi.

Obiettivo portabilità

Alcuni programmatori pensano che, implementando un'applicazione C, questa sia immediatamente portabile su tutte le piattaforme; sfortunatamente, non è sempre così. Le sezioni *Obiettivo portabilità* aiutano gli studenti a scrivere codice realmente portabile, fornendo inoltre informazioni su come il C sia in grado di raggiungere questo elevato livello di portabilità.

Ingegneria del software

Il C è un linguaggio molto efficace nell'ambito della progettazione del software, e le sezioni Ingegneria del software prendono in esame gli aspetti architetturali e progettuali che influiscono sulla realizzazione di sistemi software, specialmente nel caso di sistemi su vasta scala. Molte di queste informazioni saranno utili agli studenti nei corsi più avanzati, oltre che nel mondo del lavoro.

Esercizi di autovalutazione

Il libro propone molti esercizi corredati di risposte, così che gli studenti possano prepararsi alle esercitazioni vere e proprie. Gli studenti dovrebbero essere incoraggiati a svolgere tutti questi esercizi di autovalutazione.

Esercizi

Ogni capitolo si conclude con un insieme di esercizi di vario tipo, che permettono agli insegnanti di adattare le proprie lezioni alle esigenze particolari di ogni classe. Gli esercizi verificano l'apprendimento dei concetti e dei termini più importanti, chiedono agli studenti di scrivere singole istruzioni, piccole porzioni di funzioni, funzioni e programmi completi fino ad arrivare alla costruzione di interi progetti.

Panoramica sul libro

Capitolo 1 - Introduzione al mondo dei computer, di Internet e del World Wide Web: spiega cosa sono i computer, come funzionano e in che modo possono essere programmati. Questo capitolo introduce la programmazione strutturata e spiega perché questo insieme di tecniche abbia favorito una rivoluzione nel modo di scrivere i programmi. Il capitolo offre una breve storia dello sviluppo dei linguaggi di programmazione, dai linguaggi macchina, ai linguaggi assembly, ai linguaggi di alto livello. È discussa anche l'origine del linguaggio di programmazione C. Il capitolo include un'introduzione a un tipico ambiente di programmazione del C. Viene discusso l'enorme aumento di interesse nei confronti di Internet in seguito all'avvento del World Wide Web e del linguaggio di programmazione Java.

Capitolo 2 - Introduzione alla programmazione in C: fornisce un'introduzione sintetica ai programmi di scrittura C e offre un trattamento dettagliato delle operazioni decisionali e aritmetiche del C. Dopo aver studiato questo capitolo, lo studente avrà imparato a scrivere programmi semplici ma completi.

Capitolo 3 - Lo sviluppo di programmi strutturati in C: è probabilmente il capitolo più importante del testo, specialmente per un diligente studente di informatica. Esso introduce la nozione di algoritmi (procedure) per la risoluzione dei problemi; spiega l'importanza della programmazione strutturata per la produzione di programmi comprensibili, che siano collaudabili e modificati e che abbiano più probabilità di funzionare correttamente sin dal primo tentativo. Introduce anche i comandi di controllo fondamentali della programmazione strutturata, cioè i comandi di sequenza, selezione (`if` e `if...else`) e di iterazione (`while`). Il capitolo spiega la programmazione top-down a raffinamenti successivi, una tecnica importante per produrre in modo corretto programmi strutturati. Esso presenta un popolare strumento di aiuto alla progettazione dei programmi, ovverosia lo pseudocodice strutturato. I metodi e gli approcci usati nel Capitolo 3 sono applicabili alla programmazione strutturata in ogni linguaggio di programmazione, non solo a quella in C. Questo capitolo aiuta lo studente a sviluppare delle buone abitudini di programmazione che lo prepareranno ad affrontare gli esercizi di programmazione più consistenti del resto del libro.

Capitolo 4 - Il controllo del programma in C: raffina le nozioni di programmazione strutturata e introduce altri comandi di controllo; esamina in dettaglio l'iterazione e paragona i cicli controllati da un contatore con quelli controllati da un valore sentinella. Il comando `for` è presentato come uno strumento adatto a implementare i cicli controllati da un contatore. Sono introdotti anche il comando di selezione `switch` e quello di iterazione `do...while`. Il capitolo termina con una discussione sugli operatori logici.

Capitolo 5 - Le funzioni in C: tratta della progettazione e costruzione dei moduli del programma. Le risorse del C relative alle funzioni includono funzioni di libreria standard, funzioni definite dal programmatore, la ricorsione e la possibilità di effettuare delle chiamate per

valore. Le tecniche presentate sono essenziali per la produzione e la comprensione di programmi strutturati, specialmente di quelli più grandi, e del software che i programmatori di sistemi e di applicazioni svilupperanno più probabilmente nelle applicazioni del mondo reale. La strategia "dividi e conquista" è presentata come un mezzo efficace per risolvere i problemi più complessi, suddividendoli in componenti più semplici che interagiscono tra di loro. Agli studenti piace avere a che fare con i numeri casuali e le simulazioni e apprezzano la trattazione del gioco di dadi craps che fa un uso elegante dei comandi di controllo. In questo capitolo vengono presentate le enumerazioni, che verranno poi riprese in maggior dettaglio nel Capitolo 10. Il Capitolo 5 offre una solida introduzione alla ricorsione e include una tabella che riassume i numerosi esempi ed esercizi sulla ricorsione distribuiti nel resto del libro. Alcuni libri affrontano la ricorsione solo verso la loro fine, ma noi pensiamo che sia meglio trattare l'argomento in modo graduale nel corso di tutto il libro. Gli esaurienti esercizi includono vari problemi classici sulla ricorsione come le Torri di Hanoi.

Capitolo 6 - I vettori in C: discute la strutturazione dei dati in vettori, cioè gruppi di dati correlati e dello stesso tipo. Il capitolo presenta numerosi esempi sia sui vettori con un solo indice che su quelli con indice doppio. L'importanza della strutturazione dei dati è ampiamente riconosciuta al pari dell'utilizzo dei comandi di controllo nello sviluppo di programmi strutturati. Alcuni esempi approfondiscono varie tipiche elaborazioni dei vettori, la stampa degli istogrammi, la classificazione dei dati, il passaggio dei vettori alle funzioni e un'introduzione all'analisi dei sondaggi di opinione (con delle semplici statistiche). Un punto caratteristico di questo capitolo è un'accurata presentazione della ricerca binaria, un enorme miglioramento di quella lineare. Gli esercizi alla fine del capitolo includono una vasta selezione di problemi interessanti e stimolanti come, ad esempio, tecniche di ordinamento migliorate, la progettazione di un sistema di prenotazioni per linee aeree, un'introduzione al concetto della Turtle Graphic (resa famosa dal linguaggio LOGO) e i problemi del giro del cavallo e delle otto regine che introducono la nozione di programmazione euristica, molto utilizzata nel campo dell'intelligenza artificiale.

Capitolo 7 - I puntatori in C: presenta una delle caratteristiche più potenti del linguaggio C, ovvero, i puntatori. Il capitolo fornisce spiegazioni dettagliate sugli operatori dei puntatori, sulla chiamata per riferimento, sulle espressioni con puntatori, sull'aritmetica dei puntatori, sulla relazione tra puntatori e vettori, sui vettori di puntatori e sui puntatori a funzioni. Gli esercizi del capitolo includono una piacevole simulazione della classica gara tra la tartaruga e la lepre, vari algoritmi per il mescolamento e la distribuzione delle carte e un algoritmo ricorsivo per l'attraversamento di un labirinto. È inclusa anche una speciale sezione intitolata "Costruite il vostro computer", che spiega la programmazione in linguaggio macchina e prosegue con un progetto che comprende la progettazione e l'implementazione di un simulatore che permetta al lettore di scrivere e far funzionare i programmi sviluppati in linguaggio macchina. Questa caratteristica unica del libro sarà utile soprattutto al lettore che vorrà capire il reale funzionamento dei computer. Ai nostri studenti piace questo progetto e spesso implementano sostanziali miglioramenti; molti di questi sono proposti all'interno degli esercizi. Nel Capitolo 12, un'altra sezione speciale guiderà il lettore attraverso la costruzione di un compilatore; il linguaggio macchina prodotto dal compilatore sarà poi eseguito con il simulatore di linguaggio macchina prodotto nel Capitolo 7.

Capitolo 8 - I caratteri e le stringhe in C: tratta dei principi fondamentali dell'elaborazione dei dati non numerici. Il capitolo include una panoramica completa delle funzioni per l'elaborazione dei caratteri e delle stringhe disponibili nella libreria del C. Le tecniche discusse in

questo contesto sono ampiamente utilizzate nello sviluppo di word processor, software di impaginazione e applicazioni per l'elaborazione del testo. Allo studente piaceranno gli esercizi sui limerick, sulla generazione casuale di poesie, sulla conversione dall'inglese al pig Latino, sulla generazione di parole di sette lettere che equivalgono a un numero telefonico dato, sull'allineamento del testo, sulla protezione degli assegni, sulla scrittura della cifra di un assegno in lettere, sulla generazione del Codice Morse, sulle conversioni metriche e sulle lettere di sollecito. L'ultimo esercizio invita lo studente a utilizzare un dizionario computerizzato per creare un generatore di cruciverba.

Capitolo 9 - La formattazione dell'Input/Output in C: presenta tutte le potenti capacità di formattazione di printf e scanf. Discuteremo delle capacità di formattazione dell'output di printf, come l'arrotondamento dei valori in virgola mobile a un dato numero di cifre decimali, l'incolonnamento dei numeri, l'allineamento a destra e a sinistra, l'inserimento di informazioni letterali, la forzatura del segno positivo, la stampa degli zeri iniziali, l'uso della numerazione esponenziale, l'uso dei numeri ottali ed esadecimali e il controllo delle dimensioni di campo e della precisione. Discuteremo tutte le sequenze di escape di printf per il movimento del cursore, per la stampa dei caratteri speciali e per la produzione di un allarme acustico. Esamineremo tutte le possibilità di formattazione dell'input di scanf incluso l'input di specifici tipi di dati e come ignorare dei caratteri specifici presenti sullo stream di input. Discuteremo tutte le specifiche di conversione di scanf per leggere i numeri decimali, ottali, esadecimali, in virgola mobile, i caratteri e i valori di una stringa. Discuteremo della scansione dell'input in modo che corrisponda (o meno) ai caratteri inclusi in un gruppo di scansione. Gli esercizi del capitolo verificano virtualmente tutte le capacità di input/output formattato.

Capitolo 10 - Le strutture, le unioni, la gestione dei bit e le enumerazioni in C: presenta diverse caratteristiche importanti. Le strutture sono come i record del Pascal e di altri linguaggi: esse raggruppano dati di tipi diversi. Le strutture sono usate nel Capitolo 11 per formare file composti da record di informazioni. Nel Capitolo 12 le strutture sono usate insieme ai puntatori e all'allocazione dinamica della memoria, per formare strutture dinamiche di dati come le liste concatenate, le code, le pile e gli alberi. Le unioni consentono di utilizzare un'area di memoria per diversi tipi di dato in tempi differenti; questa condivisione può ridurre la quantità di memoria principale o secondaria richiesta da un programma. Le enumerazioni forniscono un modo conveniente per definire delle costanti simboliche utili; ciò aiuta a rendere i programmi più autoesplicativi. Le potenti capacità di manipolazione del bit del C consentono ai programmatori di scrivere programmi che usano le capacità a basso livello dell'hardware. Ciò aiuta i programmi a elaborare stringhe di bit, a impostare a on o a off i singoli bit e a immagazzinare le informazioni in modo più compatto. Tali capacità, spesso disponibili solo nei linguaggi assembly di basso livello, sono particolarmente apprezzate dai programmatori nella scrittura di software per i sistemi operativi e per la gestione delle reti. Una caratteristica di questo capitolo è la nuova versione ad alta efficienza della simulazione del mescolatore e distributore di carte. Questa è una eccellente opportunità per l'insegnante per porre l'accento sulla qualità degli algoritmi.

Capitolo 11 - L'elaborazione dei file in C: discute delle tecniche usate per elaborare i file di testo ad accesso sequenziale e casuale. Il capitolo inizia con un'introduzione alla gerarchia dei dati: dai bit ai byte, ai campi, ai record, ai file. Successivamente è presentata una semplice panoramica sui file e sugli stream del C. I file ad accesso sequenziale sono discussi usando dei programmi che mostrano come aprire e chiudere i file, come immagazzinare i dati in modo sequenziale in un file e come leggere in modo sequenziale i dati da un file. Si discute dei file

ad accesso casuale usando dei programmi che mostrano come creare un file ad accesso casuale, come leggere e scrivere i dati con accesso casuale, e come leggere i dati in modo sequenziale da un file ad accesso casuale. Il quarto programma combina molte tecniche di accesso ai file, sia sequenziali che casuali, in un programma completo per l'elaborazione di transazioni.

Capitolo 12 - Le strutture di dati in C: discute le tecniche usate per creare e manipolare delle strutture dinamiche di dati. Il capitolo inizia discutendo delle strutture ricorsive e dell'allocazione dinamica della memoria e prosegue con una discussione su come creare e conservare varie strutture dinamiche di dati come le liste concatenate, le code (o liste di attesa), le pile e gli alberi. Per ogni tipo di struttura di dati presentiamo dei programmi funzionanti e completi e mostriamo degli esempi di output. Il capitolo consente allo studente di padroneggiare i puntatori, includendo molti esempi di utilizzo dell'indirezione e della doppia indirezione (un concetto particolarmente difficile). Uno dei problemi legati all'utilizzo dei puntatori è che gli studenti hanno difficoltà a visualizzare le strutture di dati e il modo in cui i loro nodi sono collegati, così abbiamo incluso delle illustrazioni che mostrano i collegamenti e la sequenza in cui essi sono creati. L'esempio dell'albero binario è una bella pietra miliare per lo studio dei puntatori e delle strutture dinamiche di dati. Questo esempio crea un albero binario, si preoccupa dell'eliminazione dei duplicati e introduce l'attraversamento ricorsivo dell'albero in ordine anticipato, in ordine e in ordine posticipato. Gli studenti ritengono realmente esaustivo lo studio e l'implementazione di questo esempio e apprezzano in modo particolare il vedere che l'attraversamento in ordine dell'albero stampa i valori dei nodi nell'ordine stabilito. Il capitolo include un gruppo consistente di esercizi. Rilevante è la sezione speciale "Costruite il vostro compilatore". Gli esercizi accompagnano lo studente nello sviluppo di un programma di conversione dalla notazione infissa a quella polacca inversa e di un programma di valutazione di espressioni in notazione polacca inversa. Abbiamo quindi modificato l'algoritmo di valutazione di espressioni in notazione polacca inversa in modo da poter generare un codice in linguaggio macchina. Il compilatore immagazzina il suddetto codice in un file (usando le tecniche del Capitolo 11). Gli studenti eseguono quindi il codice in linguaggio macchina prodotto dai loro compilatori sul simulatore di software costruito negli esercizi del Capitolo 7!

Capitolo 13 - Il preprocessore del C: offre una discussione dettagliata delle direttive del preprocessore. Il capitolo include informazioni dettagliate sulla direttiva `#include` che induce l'inclusione di una copia di un file specificato in sostituzione della direttiva, prima che il file sia compilato, e sulla direttiva `#define` che definisce delle costanti simboliche e delle macro. Il capitolo spiega la compilazione condizionata per consentire al programmatore di controllare l'esecuzione delle direttive del preprocessore e la compilazione del codice del programma. Si parla dell'operatore # che converte il suo operando in una stringa e dell'operatore ## che concatena due token. Sono presentate le costanti simboliche `__LINE__`, `__FILE__`, `__DATE__` e `__TIME__`. Alla fine è trattata la macro `assert` del file di intestazione `assert.h`. Assert è preziosa nel collaudo, la messa a punto, la verifica e la convalida del programma.

Capitolo 14 - Argomenti avanzati: presenta vari argomenti avanzati che di solito non sono affrontati nei corsi introduttivi. Viene mostrato come redirezionare l'input di un programma in modo che provenga da un file, come redirezionare l'output in modo che sia immagazzinato in un file, come redirezionare l'output di un programma in modo che vada a costituire l'input di un altro programma (operazione chiamata piping) e come accodare l'output di un programma a un file già esistente. Viene inoltre discusso come sviluppare funzioni che usano elenchi variabili di argomenti, in che modo gli argomenti della riga di comando possano

essere passati alla funzione `main` e come possano essere utilizzati all'interno di un programma. Viene trattata la compilazione di programmi i cui componenti possono essere distribuiti in vari file, la registrazione delle funzioni con `atexit` in modo che possano essere eseguite al termine dell'esecuzione di un programma e la terminazione dell'esecuzione di un programma con la funzione `exit`. Poi si discute dei qualificatori di tipo `const` e `volatile` e viene mostrato come specificare il tipo di una costante numerica, usando dei suffissi per gli interi e i numeri in virgola mobile. Infine, si dimostra come usare la libreria per la gestione dei segnali per intercettare degli eventi inattesi, si discute della creazione e dell'uso dei vettori dinamici con le funzioni `calloc` e `realloc` e si illustra l'utilizzo delle strutture `union` come una tecnica per evitare lo spreco di spazio di memoria.

Capitolo 15 - Introduzione al C99: il C99 è uno standard emendato per il linguaggio di programmazione C che raffina ed espande le potenzialità del C89 (a cui faremo riferimento con l'espressione Standard C nei Capitoli 1-14). Tuttavia il C99 non è stato ancora largamente adottato e molti compilatori C non lo supportano completamente. Utilizzeremo l'ambiente integrato (IDE) Dev-C++ 4.9.9.2 della Bloodshed Software per illustrare le varie caratteristiche del C99.

Alcune delle nuove peculiarità del C99 includono:

- i commenti su singola linea introdotti da `//`;
- la possibilità di dichiarare una variabile in un punto qualunque prima del suo utilizzo all'interno di un blocco (compresa la clausola di inizializzazione di un comando `for`);
- gli inizializzatori designati, che permettono di inizializzare gli elementi dei vettori esplicitamente tramite i rispettivi indici e gli elementi delle `struct` e delle `union` tramite i rispettivi nomi;
- il tipo di dati `_Bool` che può rappresentare soltanto i valori 0 e 1;
- l'obbligatorietà di specificare esplicitamente un tipo di ritorno nelle funzioni (invece di assumere implicitamente il tipo di ritorno `int`);
- il supporto dei numeri complessi e dell'aritmetica complessa;
- i vettori di lunghezza variabile la cui dimensione viene stabilita a tempo di esecuzione (ma una volta fissata, non può più subire cambiamenti);
- il supporto di nomi più lunghi per gli identificatori (identificatori estesi);
- i puntatori ristretti per l'accesso riservato a regioni di memoria;
- la divisione intera affidabile;
- i vettori flessibili come membri delle strutture;
- il tipo di dati `long long int`;
- il supporto della matematica indipendente dal tipo;
- le funzioni `inline`;
- l'obbligatorietà di specificare un'espressione nel comando `return` di una funzione con un tipo di ritorno diverso da `void`;
- il divieto di utilizzare un'espressione nel comando `return` di una funzione con tipo di ritorno `void`;
- la funzione `snprintf` per prevenire errori di buffer overflow durante la composizione di stringhe in memoria.

Il nostro approccio fornisce un'introduzione al C99 e un elenco di risorse web per il lettore che voglia indagare più a fondo gli argomenti trattati.

Numerose appendici forniscono un valido materiale di riferimento. Nell'Appendice A presentiamo alcune risorse sul C disponibili su Internet e nel World Wide Web; nell'Appendice B un elenco delle risorse relative al C99 disponibili su Internet e nel World Wide Web; nell'Appendice C una tabella completa della priorità e dell'associatività degli operatori; nell'Appendice D l'insieme dei codici dei caratteri ASCII. L'Appendice E costituisce un'introduzione completa sui sistemi numerici con molti esercizi di auto-valutazione e le relative risposte. L'Appendice F fornisce una panoramica sulle Librerie Standard del C e sulle relative risorse disponibili nel Web. L'Appendice G presenta un elenco commentato di tutte le funzioni della libreria C. Infine, l'Appendice H fornisce indicazioni su come risolvere e programmare il famoso gioco del Sudoku.

Ringraziamenti

È un grande piacere ringraziare le molte persone il cui forte impegno, la cooperazione, l'amicizia e la comprensione sono stati fondamentali per la realizzazione di questo libro. In molti alla Deitel & Associates hanno dedicato molte ore a questo progetto – grazie in special modo a Abbey Deitel, Christi Kelsey e Barbara Deitel.

Desideriamo anche ringraziare due partecipanti al nostro Honors Internship program che hanno contribuito a questa pubblicazione – Alex Tuteur e Kenny Leftin, studenti di computer science rispettivamente alla Brown University e presso l'Università del Maryland.

Ci riteniamo fortunati per aver lavorato a questo progetto con l'ottimo team di professionisti dell'editore Prentice Hall. Apprezziamo gli straordinari sforzi di Marcia Horton, direttore editoriale della sezione di ingegneria e informatica della Prentice Hall. Jennifer Cappello e Dolores Mars hanno fatto un lavoro strordinario componendo il gruppo dei revisori e gestendo il processo di revisione. Francesco Santalucia (un artista indipendente) e Kristine Carney hanno realizzato una bellissima copertina. Vince O'Brien, Bob Engelhardt, Donna Crilly e Marta Samsel hanno ottimamente gestito il processo di produzione del libro.

Desideriamo ringraziare gli sforzi dei nostri revisori scientifici. In tempi molto stretti, hanno verificato il testo e i programmi, fornendo numerosi suggerimenti per migliorare l'accuratezza e la completezza della presentazione.

C Corso completo di programmazione, revisori della 4 edizione originale

John Benito (ISO/IEC JTC1 SC22 WG14 Convener)

Fred Tydeman (Tydeman Consulting, Vice-Chair of J11 (ANSI "C"))

Richard Albright (Goldey-Beacom College)

Mikhail Brikman (Salem State College)

Carol Luckhardt Redfield (St. Mary's University)

Randy Scovil (Cuesta College)

Bin Wang (Wright State University)

C Corso completo di programmazione, revisori della 5 edizione originale

Alireza Fazelpour (Palm Beach Community College)

Don Kostuch (Independent Consultant)

Ed James Beckham (Altera)

Gary Sibbitts (St. Louis Community College at Meramec)

Ian Barland (Radford University)

Kevin Mark Jones (Hewlett Packard)

Mahesh Hariharan (Microsoft)

William Mike Miller (Edison Design Group, Inc.)

Benjamin Seyfarth (Univeristy of Southern Mississippi)

William Albrecht (University of South Florida)

William Smith (Tulsa Community College)

Revisori del C99

Lawrence Jones, (UGS Corp.)

Douglas Walls (Senior Staff Engineer, C compiler, Sun Microsystems)

Desideriamo ringraziare nuovamente i revisori delle precedenti edizioni (le affiliazioni risalgono al tempo della revisione);

Rex Jaeschke (Independent Consultant; ex presidente dell'ANSI C Committee)

John Benito (Presidente del gruppo di lavoro ISO, responsabile per il linguaggio di programmazione C)

Randy Meyers (NetCom; presidente dell'ANSI C Committee; ex membro dell'ANSI C++ Committee)

Jim Brzowski (University of Massachusetts – Lowell)

Simon North (Synopsis, autore XML)

Fred Tydeman (Consulente)

Kevin Wayne (Princeton University)

Eugene Katzin (Montgomery College)

Sam Harbison (Texas Instruments, PH Author)

Chuck Allison (Tydeman Consulting)

Catherine Dwyer (Pace University)

Glen Lancaster (DePaul University)

Deena Engel (New York University)

David Falconer (California State University at Fullerton)

David Finkel (Worcester Polytechnic)

H. E. Dunsmore (Purdue University)

Jim Schmolze (Tufts University)

Geb Thomas (University of Iowa)

Gene Spafford (Purdue University)

Clovis Tondo (IBM Corporation e visiting professor alla Nova University)

Jeffrey Esakov (University of Pennsylvania)

Tom Slezak (University of California, Lawrence Livermore National Laboratory)

Gary A. Wilson (Gary A. Wilson & Assoc.; Univ. of California Berkeley Extension)

Mike Kogan (IBM Corp.; chief architect of 32-bit OS/2 2.0)

Don Kostuch (IBM Corp.; instructor in C, C++ e OOP)

Ed Lieblein (Nova University)

John Carroll (San Diego State University)

Alan Filipski (Arizona State University)

Greg Hidley (University of California, San Diego)

Daniel Hirschberg (University of California, Irvine)

Jack Tan (University of Houston)

Richard Alpert (Boston University)

Eric Bloom (Bentley College)

Vi preghiamo di inviarci i vostri commenti, suggerimenti, critiche e correzioni, al fine di migliorare ulteriormente questo libro. L'indirizzo a cui scriverci è il seguente:

```
deitel@deitel.com
```

Bene, per ora è tutto. Vi rinnoviamo il benvenuto nel mondo del C e dello sviluppo delle applicazioni del futuro. Buona fortuna e buon lavoro!

Harvey M. Deitel
Paul J. Deitel

Gli autori

Paul J. Deitel – amministratore delegato e chief technical officer della Deitel & Associates, Inc. – si è laureato presso la Sloan School of Management del MIT dove ha studiato tecnologia dell'informazione. Con la Deitel & Associates ha realizzato dei corsi di formazione su Java, C, C++, Java e C# per clienti di primaria importanza, quali IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA (Kennedy Space Center), National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, Invensys e molte altre organizzazioni. Ha tenuto conferenze sui linguaggi C++ e Java per il Boston Chapter dell'Association for Computing Machinery. Paul e suo padre Harvey M. Deitel sono autori di libri di informatica che hanno riscosso grande successo in tutto il mondo.

Harvey M. Deitel – presidente e chief strategy officer della Deitel & Associates, Inc. – si occupa di informatica da oltre 40 anni e ha maturato una significativa esperienza accademica e aziendale. È uno dei più affermati istruttori di informatica a livello mondiale. Ha conseguito la laurea in scienze presso il Massachusetts Institute of Technology (MIT) e il dottorato presso l'Università di Boston. Ha insegnato per 20 anni nelle università ed è stato rettore della facoltà di informatica del Boston College, prima di fondare la Deitel & Associates, Inc. con il figlio Paul J. Deitel. Ha partecipato a progetti d'avanguardia dei sistemi operativi con memoria virtuale presso l'IBM e il MIT, sviluppando quelle tecniche che oggi sono ampiamente implementate nei sistemi UNIX, Linux e Windows NT. È autore e coautore con il figlio Paul J. di numerosi volumi e di package multimediali e ne sta scrivendo altri. I testi di Harvey M. Deitel hanno ottenuto un riconoscimento internazionale e sono stati tradotti e pubblicati in giapponese, tedesco, cinese, russo, italiano, francese, spagnolo, coreano, polacco, portoghese, greco, urdu e turco. Harvey M. Deitel ha tenuto numerosi seminari professionali per grandi aziende, istituzioni universitarie e organizzazioni governative e militari.

Deitel & Associates, Inc.

Deitel & Associates, Inc. è una società di fama internazionale specializzata nella formazione aziendale e nella realizzazione di testi sui linguaggi di programmazione, sulle tecnologie Internet e del World Wide Web, sulle tecnologie orientate agli oggetti e sullo sviluppo di applicazioni e-business ed e-commerce. Deitel & Associates è membro del *World Wide Web Consortium* (*W3C*). La società ha realizzato corsi introduttivi e avanzati sui pricipali linguaggi e piattaforme di programmazione: Java, C, C++, C#, Visual C++, Visual Basic, XML, Perl, Python e sulle tecnologie orientate agli oggetti, sulla programmazione di Internet e del World Wide Web.

I fondatori della Deitel & Associates, Inc. sono Harvey M. Deitel e Paul J. Deitel. Fra i clienti della società figurano molte delle più importanti imprese mondiali, enti governativi e delle forze armate USA e altre organizzazioni. Grazie alla trentennale collaborazione con la società editrice Prentice Hall, la Deitel & Associates pubblica testi d'avanguardia sulla programmazione, libri per l'informatica professionale, realizza corsi multimediali interattivi e corsi di addestramento via Web; inoltre crea contenuti per famose piattaforme di e-learning come WebCT, Blackboard e Pearson's CourseCompass. La Deitel & Associates e gli autori possono essere contattati via e-mail al seguente indirizzo:

```
deitel@deitel.com
```

Per avere maggiori informazioni sulla Deitel & Associates e per consultare l'elenco aggiornato delle sue pubblicazioni e dei corsi di addestramento aziendale, visitate il sito Web:

```
www.deitel.com
```

Capitolo 1

Introduzione ai computer, a Internet e al Web

Obiettivi

- Comprendere le nozioni elementari relative ai computer.
- Familiarizzare con i diversi tipi di linguaggi di programmazione.
- Conoscere la storia del linguaggio di programmazione C.
- Venire a conoscenza della libreria standard del C.
- Comprendere gli elementi dell'ambiente di sviluppo tipico dei programmi C.
- Apprezzare le ragioni per le quali è appropriato apprendere il C in un primo corso di programmazione.
- Apprezzare le ragioni per le quali il C fornisce le basi fondamentali per i successivi studi sulla programmazione, in generale, e sul C++, Java e C# in particolare.
- Conoscere la storia di Internet e del World Wide Web.

1.1 Introduzione

Benvenuti nel mondo del C! Abbiamo lavorato duramente per creare quella che, speriamo sinceramente, sarà per voi una esperienza informativa e divertente. Questo libro è unico tra quelli sul C per i seguenti motivi:

- È adatto a chi ha una conoscenza tecnica, ma ha poca o nessuna esperienza di programmazione.
- È adatto ai programmatori esperti che desiderano una trattazione rigorosa e approfondita del linguaggio.

Come può un libro piacere a entrambi i gruppi? La risposta è che il libro enfatizza il raggiungimento della *chiarezza* del programma, attraverso le provate tecniche di "programmazione strutturata". Quelli che non sono ancora dei programmatori impareranno sin dall'inizio il modo "giusto". Abbiamo tentato di scrivere in un modo chiaro e diretto. Il libro è abbondantemente illustrato. Cosa probabilmente anche più importante, il libro presenta un enorme numero di programmi C funzionanti e mostra gli output (risultati) prodotti, quando questi programmi sono eseguiti su un computer. Ci riferiamo a questa caratteristica chiamandola approccio "live-code". È possibile scaricare tutti i programmi di esempio direttamente dal sito Web `www.deitel.com` o dal sito `www.apogeonline.com/libri/88-503-2254-2/scheda`.

I primi quattro capitoli introducono i principi della elaborazione elettronica, la programmazione dei computer e il linguaggio C. I principianti che hanno seguito i nostri corsi ci hanno assicurato che il materiale di questi capitoli fornisce una solida base per l'approfondimento del C sviluppato nei Capitoli dal 5 al 14. Di solito, i programmatori esperti leggeranno velocemente i primi quattro capitoli e scopriranno in seguito che il C è trattato nei Capitoli dal 5 al 14 in modo rigoroso e, allo stesso tempo, interessante. Essi apprezzeranno in modo particolare i capitoli avanzati, con la loro trattazione dettagliata dei puntatori, delle stringhe, dei file e delle strutture di dati.

Molti programmatori esperti ci hanno riferito il loro apprezzamento per la nostra discussione sulla programmazione strutturata. Per quanto avessero programmato spesso in un linguaggio strutturato come il Pascal, non erano in grado di scrivere il miglior codice possibile perché non avevano mai conosciuto in modo formale la programmazione strutturata. Ora, dopo avere imparato il C da questo libro, hanno la possibilità di migliorare il proprio stile di programmazione. Per cui, che siate dei principianti o dei programmatori esperti, in questo libro ci sono molte cose che potranno informarvi, divertirvi e stimolarvi.

Molti hanno familiarità con le cose interessanti che i computer sono in grado di fare. In questo corso, imparerete a comandare i computer perché facciano quelle cose. È il *software* (ovverosia, le istruzioni che voi scrivete per ordinare al computer di eseguire delle azioni e prendere delle decisioni) che controlla i computer (detti spesso *hardware*). Questo testo fornisce una introduzione alla programmazione dell'ANSI C, la versione standardizzata nel 1989 negli Stati Uniti dall'Istituto Nazionale Americano per gli Standard (ANSI) e, nel resto del mondo, dall'Organizzazione Internazionale per gli Standard (ISO). Chiameremo tale linguaggio Standard C. Tratteremo anche il C99, ovvero, l'ultima versione dello standard C. Il C99 non è stato ancora largamente adottato; quindi abbiamo scelto di trattarlo nel Capitolo 15, mentre nei Capitoli 1-14 parleremo dello Standard (ANSI/ISO) C.

L'utilizzo dei computer sta crescendo in quasi tutti i campi delle attività umane. In un'era di costi costantemente in crescita, quelli della elaborazione elettronica sono diminuiti vertiginosamente, a causa dei rapidissimi sviluppi della tecnologia hardware e software. Quei computer che, solo una ventina di anni fa, potevano riempire grandi stanze e costavano milioni di dollari, oggi possono essere prodotti su lamine di silicio (chip) più piccole di un'unghia, che costano forse pochi dollari. Ironicamente, il silicio è uno dei materiali più abbondanti sulla terra: è un ingrediente della sabbia comune. La tecnologia del chip di silicio ha reso l'elaborazione elettronica così economica che, in tutto il mondo, sono in uso centinaia di milioni di computer general-purpose (per scopi generici), aiutando la gente negli affari, nell'industria, nel governo e nelle loro vite personali, che potrebbero facilmente raddoppiarsi in una manciata di anni.

Al giorno d'oggi il C++ e Java, due linguaggi orientati agli oggetti che si basano sul C, stanno attirando molto interesse e nel mercato dei linguaggi di programmazione molti dei fornitori principali, piuttosto che offrire dei prodotti separati, ora vendono semplicemente un C/C++ combinato. Ciò dà all'utente, se lo desidera, la possibilità di continuare a programmare in C e di migrare gradualmente al C++, quando lo riterrà opportuno.

State dunque per intraprendere un percorso interessante e, speriamo, remunerativo. Man mano che procederete, se vorrete comunicare con noi, inviateci una email su Internet all'indirizzo `deitel@deitel.com` o esplorando il nostro sito Web all'indirizzo `www.deitel.com`. Faremo tutto il possibile per rispondervi velocemente. Speriamo che vi divertirete imparando il C con questo libro.

1.2 Che cosa è un computer?

Un *computer* è un dispositivo in grado di eseguire dei calcoli e di prendere delle decisioni logiche, il tutto a una velocità superiore di milioni e anche miliardi di volte a quella degli esseri umani. Oggi per esempio, molti personal computer possono eseguire decine di milioni di addizioni per secondo. Una persona che operi a una calcolatrice da tavolo potrebbe aver bisogno di una vita intera, per completare lo stesso numero di calcoli che un personal computer può eseguire in un secondo. (Punto di riflessione: come potreste sapere se la persona ha sommato correttamente i numeri? Come potreste sapere se il computer ha sommato correttamente i numeri?). I *supercomputer* più veloci oggi possono eseguire centinaia di miliardi di addizioni per secondo. E nei laboratori di ricerca stanno già funzionando computer in grado di eseguire migliaia di miliardi di istruzioni per secondo.

I computer elaborano i *dati* sotto il controllo di insiemi di istruzioni chiamati *programmi per computer*. Tali programmi guidano il computer per mezzo di insiemi ordinati di azioni specificati da persone chiamate *programmatori di computer*.

Un computer è composto da vari dispositivi (come la tastiera, lo schermo, il mouse, i dischi, la memoria, i lettori di DVD e CD-ROM e le unità di elaborazione) che sono chiamati complessivamente *hardware*. I programmi che possono essere eseguiti su un computer sono chiamati *software*. Il costo dell'hardware è diminuito vertiginosamente negli ultimi anni, al punto che il personal computer è diventato un oggetto di uso comune. Sfortunatamente, per decenni i costi di sviluppo del software sono andati aumentando costantemente man mano che i programmatori hanno sviluppato applicazioni sempre più potenti e complesse, senza miglioramenti significativi nella tecnologia di sviluppo del software. In questo libro, imparerete dei metodi di sviluppo del software di provata efficacia che aiutano nell'attività di controllo di organizzazioni e addirittura riducono i costi dello sviluppo del software: la *programmazione strutturata*, la *programmazione top-down per raffinamenti successivi*, la *programmazione modulare*.

1.3 L'organizzazione del computer

Indipendentemente dalle differenti apparenze fisiche, ogni computer può essere virtualmente concepito come suddiviso in sei *unità logiche* o sezioni:

1. *Unità di input (ingresso di dati)*. Questa è la sezione "ricevente" del computer. Essa ottiene informazioni (dati e programmi per il computer) dai *dispositivi di input* e mette queste informazioni a disposizione delle altre unità, in modo che possano essere elaborate. La maggior parte delle informazioni oggigiorno è immessa nei computer attraverso una tastiera e un mouse. E' anche possibile inserire informazioni per mezzo del riconoscimento vocale, da immagini acquisite tramite scanner e ricevendo dei dati da una rete come Internet.
2. *Unità di output (uscita di dati)*. Questa è la sezione di "spedizione" del computer. Essa prende l'informazione che è stata elaborata dal computer e la invia a diversi *dispositivi di output*, in modo da renderla disponibile per l'utilizzo all'esterno del computer. Oggigiorno la maggior parte delle informazioni è inviata all'esterno dei computer attraverso la visualizzazione sullo schermo, la stampa su carta oppure per controllare altri dispositivi. I computer possono anche inviare le loro informazioni a delle reti come Internet.

3. *Unità di memoria.* Questa è la sezione di "magazzino" con accesso rapido e capacità relativamente bassa del computer. Essa conserva le informazioni che sono state immesse attraverso l'unità di input, in modo che possano essere rese immediatamente disponibili quando saranno necessarie. L'unità di memoria conserva anche le informazioni che sono già state elaborate, fintanto che possano ancora essere inviate ai dispositivi di output dalla corrispondente unità. L'unità di memoria è spesso chiamata anche *memoria* o *memoria primaria.*
4. *Unità aritmetica e logica (ALU).* Questa è la sezione di "produzione" del computer. È responsabile dell'esecuzione dei calcoli come le somme, le sottrazioni, le moltiplicazioni e le divisioni. Essa contiene i meccanismi di decisione che consentono al computer, per esempio, di confrontare due elementi prelevati dalla unità di memoria, per determinare se sono uguali.
5. *Unità di elaborazione centrale (CPU).* Questa è la sezione "amministrativa" del computer. È il coordinatore del computer ed è responsabile della supervisione per le operazioni delle altre sezioni. La CPU indica alla unità di input il momento in cui le informazioni dovranno essere lette nella unità di memoria, indica alla ALU, il momento in cui le informazioni della memoria dovranno essere utilizzate per i calcoli, e indica alla unità di output il momento in cui dovrà inviare le informazioni dalla unità di memoria a certi dispositivi di output. Molti computer al giorno d'oggi hanno diverse unità di elaborazione centrale e quindi possono portare a termine molte operazioni contemporaneamente; tali computer sono chiamati *multiprocessore.*
6. *Unità di memoria secondaria.* Questa è la sezione di "magazzino" a lungo termine e con alta capacità del computer. I programmi e i dati, che in un certo momento non sono utilizzati dalle altre unità, saranno normalmente sistemati su un dispositivo di memoria secondaria (come i dischi), finché non saranno nuovamente necessari; probabilmente ore, giorni, mesi o anche anni più tardi. I tempi di accesso alle informazioni memorizzate nella memoria secondaria sono molto più alti di quelli necessari per accedere alle informazioni memorizzate nelle memoria primaria; tuttavia il costo unitario della memoria secondaria è molto minore di quello della memoria primaria.

1.4 L'evoluzione dei sistemi operativi

I primi computer erano in grado di eseguire soltanto un *job (processo)* o un *task (lavoro)* alla volta. Questo modo di operare dei computer è spesso chiamato *elaborazione batch (elaborazione a lotti)* monoutente. Il computer esegue un solo programma per volta, mentre i dati sono elaborati in gruppi ossia batch. In questi primi sistemi, generalmente, gli utenti sottoponevano i loro processi al centro di elaborazione in mazzi di schede perforate. Gli utenti dovevano spesso aspettare ore, o anche giorni, prima che i tabulati fossero restituiti alle loro scrivanie.

Per rendere più agevole l'utilizzo dei computer, venne sviluppato un software particolare chiamato sistema operativo. I primi sistemi operativi avevano il compito di gestire scorrevolmente la transizione fra i vari job. Ciò rendeva minimo il tempo necessario agli operatori per passare da un job all'altro, aumentando così la quantità di lavoro, o velocità di elaborazione, che i computer potevano sostenere.

Man mano che i computer sono diventati più potenti, si è reso evidente che l'elaborazione batch monoutente utilizzava raramente in modo efficiente le risorse del computer, in

quanto la maggior parte del tempo veniva spesa aspettando il completamento del lavoro dei lenti dispositivi di input/output. Si pensò quindi di fare in modo che molti processi o lavori potessero *condividere (share)* le risorse del computer per ottenerne un miglior utilizzo. Questo modo di operare è detto *multiprogrammazione*. La multiprogrammazione richiede l'elaborazione "simultanea" di molti processi sul computer: questo, in altre parole, condivide le proprie risorse tra i vari processi che competono per la sua attenzione. Con i primi sistemi operativi in multiprogrammazione però, gli utenti dovevano ancora sottoporre i processi su mazzi di schede perforate e attendere ore o giorni per i risultati.

Negli anni '60, molti gruppi industriali e le università concepirono i sistemi operativi con *timesharing (condivisione del tempo)*. Il timesharing è un caso speciale di multiprogrammazione per mezzo del quale gli utenti accedono al computer attraverso dei *terminali*, ovvero, dei dispositivi dotati di tastiera e schermo. In un tipico sistema di computer in timesharing, potrebbero esserci dozzine o anche centinaia di utenti che condividono contemporaneamente il computer. In realtà, il computer non serve tutti gli utenti in modo simultaneo. Esso esegue piuttosto una piccola porzione del processo di un utente e in seguito presta il proprio servizio all'utente successivo. Il computer fa tutto ciò così velocemente che, in un secondo, è in grado di fornire molte volte il proprio servizio a ogni utente. In questo modo i programmi degli utenti *sembrano* girare simultaneamente. Un vantaggio del timesharing è che l'utente riceve una risposta alla sua richiesta in modo pressoché immediato invece di dover aspettare lunghi intervalli di tempo come avveniva con i precedenti modelli di elaborazione.

1.5 I personal computer, i sistemi distribuiti e i sistemi client/server

Nel 1977, la Apple Computer rese popolare il fenomeno della *elaborazione personale*. Al principio, questa era il sogno di ogni hobbista. I computer divennero abbastanza economici perché la gente potesse comprarli per il proprio uso personale o per i propri affari. Nel 1981, la IBM, il maggior fornitore di computer nel mondo, introdusse il Personal Computer IBM. In seguito a ciò rapidamente l'elaborazione personale diventò legittima negli affari, nell'industria e nelle organizzazioni governative.

Questi computer erano però delle unità "indipendenti": la gente eseguiva il proprio lavoro sulla propria macchina e quindi trasportava avanti e indietro dei dischi, per condividere le informazioni. Per quanto i primi personal computer non fossero abbastanza potenti da gestire molti utenti in timesharing, queste macchine potevano però essere collegate insieme in reti di computer, a volte su linee telefoniche e a volte in reti locali (LAN: Local Area Network) all'interno di una organizzazione. Ciò ha portato al fenomeno della *elaborazione distribuita*, in cui il carico di elaborazione di una organizzazione, invece di essere eseguito necessariamente in qualche installazione centrale di computer, è distribuito attraverso le reti alle varie postazioni in cui viene svolto effettivamente il lavoro. I personal computer erano sufficientemente potenti da gestire le richieste di elaborazione di utenti individuali e le fondamentali attività della comunicazione: passare da una macchina all'altra le informazioni in forma elettronica.

Oggigiorno, i personal computer più potenti lo sono tanto quanto le macchine da un milione di dollari di solo dieci-quindici anni fa. Le macchine desktop (da tavolo) più potenti (dette *workstation*) forniscono ai singoli utenti delle capacità enormi. L'informazione è facilmente condivisa attraverso le reti di computer, nelle quali alcune macchine, dette *file server*

(fornitori di servizi per i file), mettono a disposizione un magazzino comune di dati che può essere utilizzato da macchine *client (clienti)* distribuite in tutta la rete; da cui il termine *elaborazione client/server*. Il C, il C++ e Java sono diventati i linguaggi di programmazione scelti per scrivere il software dei sistemi operativi, delle reti di computer e delle applicazioni distribuite in ambienti client/server. Oggi i più diffusi sistemi operativi come Unix, Linux, Mac OS X (pronunciato "OS ten") e Windows mettono a disposizione il genere di caratteristiche discusse in questa sezione.

1.6 I linguaggi macchina, assembly e di alto livello

I programmatori scrivono le istruzioni in vari linguaggi di programmazione, alcuni direttamente comprensibili dal computer e altri che richiedono dei passi intermedi di *traduzione*. Oggigiorno si utilizzano centinaia di linguaggi per i computer. Questi possono essere suddivisi in tre categorie generali:

1. Linguaggi macchina
2. Linguaggi assembly
3. Linguaggi di alto livello

Ogni computer può comprendere direttamente soltanto il proprio *linguaggio macchina*. Il linguaggio macchina è la "lingua naturale" di un particolare computer. Esso è determinato dalla progettazione dell'hardware del computer. I linguaggi macchina consistono generalmente di sequenze di numeri (riducibili in definitiva a successioni di 1 e 0) che ordinano ai computer di eseguire, una per volta, le loro operazioni più elementari. I linguaggi macchina *dipendono dalla macchina* (in altre parole, un particolare linguaggio macchina può essere utilizzato soltanto su un tipo di computer). Tali linguaggi sono scomodi per gli esseri umani, come si può vedere dal seguente frammento di programma in linguaggio macchina, che aggiunge la paga degli straordinari a quella base e immagazzina il risultato nella paga lorda.

```
+1300042774
+1400593419
+1200274027
```

La programmazione in linguaggio macchina era semplicemente troppo lenta e noiosa per la maggior parte dei programmatori. Invece di usare le sequenze di numeri che i computer potevano capire direttamente, i programmatori cominciarono a usare delle abbreviazioni simili all'inglese, per rappresentare le operazioni elementari. Tali abbreviazioni formarono le basi per i *linguaggi assembly*. Furono sviluppati dei *programmi traduttori*, chiamati *assembler (assemblatori)*, per convertire in linguaggio macchina, alla velocità del computer, i programmi scritti in linguaggio assembly. Anche il seguente frammento di programma in linguaggio assembly aggiunge la paga degli straordinari a quella base e immagazzina il risultato nella paga lorda, ma in modo più chiaro del suo equivalente in linguaggio macchina:

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

Nonostante tale codice sia più chiaro per gli esseri umani, esso risulta incomprensibile ai computer finché non venga tradotto in linguaggio macchina.

L'utilizzo dei computer aumentò rapidamente con l'avvento dei linguaggi assembly, ma programmare in questi linguaggi richiedeva ancora molte istruzioni per eseguire anche il più

semplice dei compiti. Per accelerare il processo di programmazione, furono sviluppati dei *linguaggi di alto livello* in cui si poteva scrivere una singola istruzione per eseguire un compito essenziale. I programmi traduttori, che convertono in linguaggio macchina il codice scritto in quello di alto livello, sono chiamati *compilatori*. I linguaggi di alto livello consentono ai programmatori di scrivere delle istruzioni, che sembrano quasi simili all'inglese di ogni giorno e contengono le notazioni matematiche utilizzate comunemente. Un programma per la busta paga scritto in un linguaggio di alto livello potrebbe contenere una istruzione come:

```
grossPay = basePay + overTimePay
```

Ovviamente dal punto di vista dei programmatori, i linguaggi di alto livello sono molto più desiderabili, in confronto ai linguaggi macchina o a quelli assembly. Il C, il C++ e Java sono tra i più potenti e più diffusamente utilizzati linguaggi di programmazione di alto livello.

La compilazione di un programma scritto in un linguaggio ad alto livello in linguaggio macchina può richiedere un tempo di elaborazione considerevole. I programmi *interprete* furono sviluppati per eseguire direttamente i programmi scritti in linguaggi ad alto livello senza la necessità di compilare questi ultimi in linguaggio macchina. Nonostante i programmi compilati vengano eseguiti molto più velocemente di quelli interpretati, gli interpreti hanno successo negli ambienti di sviluppo in cui i programmi vengono ricompilati frequentemente non appena vengono aggiunte nuove caratteristiche e corretti degli errori. Una volta che il programma è stato sviluppato, può esserne prodotta una versione compilata affinché venga eseguita in modo più efficiente.

1.7 FORTRAN, COBOL, Pascal e Ada

Sono state sviluppate centinaia di linguaggi ad alto livello, ma soltanto alcuni hanno ottenuto un largo consenso. Il *FORTRAN* (FORmula TRANslator, Traduttore di formula) fu sviluppato dalla IBM negli anni '50 per essere utilizzato nelle applicazioni scientifiche e di ingegneria che richiedono complessi calcoli matematici. Il FORTRAN è utilizzato diffusamente ancora oggi, soprattutto nelle applicazioni di ingegneria.

Il *COBOL* (COmmon Business Oriented Language, Linguaggio orientato alle comuni attività commerciali) fu sviluppato nel 1959 da produttori di computer e da utenti governativi e industriali. Il COBOL è utilizzato per le applicazioni commerciali che richiedono una precisa ed efficiente manipolazione di grandi quantità di dati. Una porzione notevole del software commerciale è ancora programmato in COBOL. Durante gli anni '60, molti grandi progetti di sviluppo di software incontrarono delle serie difficoltà. Si era tipicamente in ritardo rispetto ai piani di sviluppo del software, i costi eccedevano notevolmente i budget e i prodotti finiti non erano affidabili. La gente cominciò a comprendere che lo sviluppo del software era una attività più complessa di quanto avesse immaginato. L'attività di ricerca degli anni '60 ebbe come risultato lo sviluppo della *programmazione strutturata*: un approccio disciplinato alla scrittura di programmi che siano più chiari di quelli non strutturati, più facili da testare e correggere e più facili da modificare.

Uno dei risultati più tangibili prodotti da questa ricerca fu lo sviluppo, da parte del Professor Nicklaus Wirth nel 1971, del linguaggio di programmazione Pascal. Il Pascal, che prese il suo nome dal matematico e filosofo del diciassettesimo secolo Blaise Pascal, fu progettato per insegnare la programmazione strutturata e divenne rapidamente il linguaggio di programmazione preferito nella maggior parte delle università. Sfortunatamente, il linguaggio mancava di molte delle strutture necessarie per renderlo utile nelle applicazioni commerciali, industriali e governative, perciò non fu accettato diffusamente in questi ambienti.

Il linguaggio di programmazione *Ada* fu sviluppato sotto il patrocinio del Dipartimento della Difesa degli Stati Uniti (U.S. Department of Defense: DOD), durante gli anni '70 e all'inizio degli '80. Per produrre gli imponenti sistemi software di comando e di controllo del DOD, erano stati utilizzati centinaia di linguaggi diversi. Il DOD voleva un unico linguaggio che potesse andare incontro alla maggior parte delle proprie necessità. Il linguaggio prese il suo nome da Lady Ada Lovelace, figlia del poeta Lord Byron. A Lady Lovelace è attribuito il merito di avere scritto, agli inizi del XIX secolo, il primo programma per computer del mondo (per la Macchina Analitica, un dispositivo meccanico per l'elaborazione progettato da Charles Babbage). Una importante caratteristica dell'Ada è il *multitasking* che consente ai programmatori di specificare che molte attività dovranno verificarsi in parallelo. Alcuni linguaggi di alto livello utilizzati diffusamente, di cui abbiamo discusso (inclusi il C e il C++), consentono di scrivere programmi che eseguono soltanto una attività per volta. Java, attraverso una tecnica chiamata *multithreading*, consente ai programmatori di scrivere programmi con attività in parallelo.

1.8 La storia del C

Il C si è evoluto da due precedenti linguaggi: il BCPL e il B. Il BCPL fu sviluppato da Martin Richards, nel 1967, come un linguaggio per scrivere il software dei sistemi operativi e dei compilatori. Ken Thompson prese a modello il BCPL, per molte caratteristiche del suo linguaggio B, e usò B per creare le prime versioni del sistema operativo UNIX nei Bell Laboratories, nel 1970, su un computer DEC PDP-7. Il BCPL e il B erano linguaggi "non tipizzati": ogni unità di informazione occupava una "word" (parola) nella memoria e l'onere di assegnare un tipo di dato alle variabili ricadeva sulle spalle del programmatore.

Il linguaggio C fu una evoluzione del B sviluppata da Dennis Ritchie, nei Bell Laboratories, e fu implementato originariamente su un computer DEC PDP-11, nel 1972. Il C utilizza molti importanti concetti del BCPL e del B, mentre aggiunge la tipizzazione dei dati e altre potenti caratteristiche. Il C inizialmente divenne famoso come il linguaggio in cui era stato sviluppato il sistema operativo UNIX. Oggi, sono scritti in C e/o in C++ quasi tutti i principali sistemi operativi dell'ultima generazione. Il C è disponibile sulla maggior parte dei computer. Il C è indipendente dall'hardware. Con una progettazione attenta, è possibile scrivere in C dei programmi che siano *portabili* sulla maggior parte dei computer.

Dagli ultimi anni '70, il C si è evoluto in quello che oggi è chiamato "C tradizionale". La pubblicazione, nel 1978, del libro di Kernighan e Ritchie, *Il linguaggio di programmazione C*, ha attirato molte attenzioni su questo linguaggio. Questa pubblicazione è diventata, in campo informatico, uno dei libri di maggior successo di tutti i tempi.

La rapida espansione del C sui diversi tipi di computer (detti, a volte, *piattaforme hardware*) ha prodotto molte varianti, che erano simili, ma spesso incompatibili. Ciò rappresentava un problema serio per i programmatori, che avevano bisogno di sviluppare un codice che potesse girare su piattaforme diverse. Divenne allora evidente che era necessaria una versione standard del C. Nel 1983, per "fornire una definizione del linguaggio che non fosse ambigua e che fosse indipendente dalle macchine", si creò il comitato tecnico X3J11, alle dipendenze del Comitato Nazionale Americano per gli Standard dei Computer e l'Elaborazione della Informazione (X3). Lo standard fu approvato nel 1989 e fu aggiornato nel 1999. Il documento degli standard si chiama INCITS/ISO/IEC 9899-1999. Copie di questo documento possono essere richieste all'Istituto Nazionale Americano per gli Standard (`www.ansi.org`) all'indirizzo `webstore.ansi.org/ansidocstore`.

Il C99 è uno standard emendato per il linguaggio di programmazione C che raffina ed espande le potenzialità di quest'ultimo. Il C99 non è stato ancora largamente adottato e non tutti i principali compilatori lo supportano; tra questi, molti supportano solo un sottoinsieme del linguaggio. I Capitoli 1-14 di questo libro sono fondati sullo Standard (ANSI/ISO) C largamente adottato a livello internazionale, mentre il Capitolo 15 introduce il C99 e fornisce i collegamenti ai più diffusi compilatori ed ambienti integrati (IDE) che lo supportano.

Obiettivo portabilità 1.1

Dato che il C è un linguaggio indipendente dagli hardware ed è largamente diffuso, le applicazioni scritte in C possono essere eseguite, con poche o nessuna modifica, su una vasta gamma di sistemi di computer differenti.

[*Nota*: includeremo molti di questi *Obiettivi portabilità* per evidenziare tecniche che vi aiuteranno a scrivere programmi che possano funzionare su diversi computer, con piccole modifiche o nessuna modifica. Metteremo in evidenza anche le *Buone abitudini* (abitudini che possono aiutarvi a scrivere programmi che siano chiari, comprensibili, mantenibili e facili da testare e correggere, cioè, ripulire dagli errori), gli *Errori tipici* (problemi da cui essere messi in guardia in modo che non commettiate gli stessi errori nei vostri programmi), gli *Obiettivi efficienza* (tecniche che vi saranno di aiuto per scrivere programmi che girino più velocemente e che usino una minor quantità di memoria), i consigli di *Collaudo e messa a punto* (tecniche che vi aiuteranno a eliminare gli errori dai vostri programmi e, cosa più importante, tecniche che vi aiuteranno a scrivere programmi senza errori al primo colpo) e le osservazioni di *Ingegneria del software* (concetti che influiscono sull'architettura complessiva e sulla qualità di un sistema software, specialmente nel caso di sistemi su vasta scala). Molte di queste tecniche e abitudini sono soltanto delle linee guida; senza dubbio, svilupperete un vostro stile preferito di programmazione.]

1.9 La libreria standard del C

I programmi scritti in C, come apprenderete nel Capitolo 5, consistono di moduli o pezzi chiamati *funzioni*. Voi potrete programmare tutte le funzioni di cui avrete bisogno per scrivere un programma C, ma la maggior parte dei programmatori C traggono vantaggio da una ricca collezione di funzioni già esistenti, chiamata *libreria standard del C*. Di conseguenza, ci sono in realtà due aspetti di come programmare in C da imparare. Il primo è imparare il linguaggio C in sé, mentre il secondo è imparare a utilizzare le funzioni della libreria standard del C. Nel corso di questo libro, parleremo di molte di quelle funzioni. Il libro di P.J. Plauger *The Standard C Library* dovrebbe essere letto dai programmatori che abbiano bisogno di una profonda conoscenza delle funzioni incluse nella libreria, di come implementarle e di come usarle per scrivere un codice portabile.

Questo libro incoraggia a utilizzare un *approccio di costruzione a blocchi* per creare i programmi. Evitate di inventare nuovamente la ruota e utilizzate i pezzi già esistenti: questa si chiama *riusabilità del software*. Quando programmerete in C, userete tipicamente i seguenti blocchi:

- Le funzioni della libreria standard.
- Le funzioni create da voi stessi.
- Le funzioni che sono state sviluppate da altre persone e che vi sono state messe a disposizione.

Il vantaggio di creare le vostre funzioni è che saprete esattamente in che modo funzionino. Sarete in grado di esaminare il codice C. Lo svantaggio è dato dal dispendio di tempo richiesto dallo sforzo per progettare e sviluppare delle nuove funzioni.

Usare le funzioni già esistenti vi eviterà di inventare nuovamente la ruota. Nel caso delle funzioni dello standard ANSI, saprete che sono state scritte con molta cura e che i vostri programmi avranno un'alta probabilità di essere portabili, poiché state utilizzando delle funzioni che sono virtualmente disponibili su tutte le implementazioni ANSI C.

Obiettivo efficienza 1.1

Utilizzare le funzioni della libreria standard ANSI, invece di scrivere le vostre corrispondenti versioni, potrà migliorare l'efficienza del programma, perché quelle funzioni sono state scritte con cura per una esecuzione efficiente.

Obiettivo portabilità 1.2

Utilizzare le funzioni della libreria standard ANSI, invece di scrivere le vostre corrispondenti versioni, potrà migliorare la portabilità del programma, perché quelle funzioni sono state incluse in quasi tutte le versioni del C.

1.10 Il C++

Il C++ è un'estensione del C sviluppata da Bjarne Stroustrup presso i Bell Laboratories. Il C++ fornisce una serie di caratteristiche che rendono "più attraente" il linguaggio C. La cosa più importante è che fornisce delle risorse per la *programmazione orientata agli oggetti*. Il C++ è diventato un linguaggio dominante sia nel campo industriale che in quello accademico.

Gli *oggetti* sono essenzialmente dei *componenti* software riusabili che rappresentano elementi del mondo reale. C'è una rivoluzione in atto nell'industria del software. Sviluppare del software velocemente, senza errori e in modo economico rimane un obiettivo sfuggente, in un periodo in cui le richieste di software nuovo e sempre più potente stanno aumentando in maniera considerevole.

Gli sviluppatori di software si stanno rendendo conto che usare un approccio alla progettazione e all'implementazione modulare e orientato agli oggetti può contribuire a rendere i gruppi che sviluppano software molto più produttivi di quanto sia possibile tramite le tecniche di programmazione convenzionali.

Molti ritengono che oggi il miglior percorso formativo sia imparare a padroneggiare il C per poi apprendere il C++.

1.11 Java

Molte persone pensano che il prossimo campo in cui i microprocessori avranno un profondo impatto sarà quello dei dispositivi elettronici intelligenti per il mercato di massa. Sun Microsystems, prendendo atto di ciò, fondò nel 1991 un progetto di ricerca interno alla propria corporazione dal nome in codice Green. Il progetto diede come risultato lo sviluppo di un linguaggio basato sul C e sul C++ che il suo creatore, James Gosling, chiamò Oak per via di un albero di quercia visibile dalla sua finestra alla Sun. In seguito si scoprì che esisteva già un linguaggio per computer chiamato Oak. In seguito a una visita di un gruppo di persone della Sun a un caffè locale, fu proposto il nome *Java* e tale nome rimase in uso.

Tuttavia il progetto Green incontrò qualche difficoltà. Il mercato dei dispositivi elettronici intelligenti non si stava sviluppando così velocemente come Sun aveva previsto. Ancora peggio, un importante contratto per il quale Sun era stata in competizione fu assegnato a un'altra azienda, mettendo il progetto a rischio di essere cancellato. Per pura buona fortuna il World Wide Web divenne molto popolare nel 1993 e Sun riconobbe l'immediato potenziale di utilizzare Java per creare pagine Web dal cosiddetto *contenuto dinamico*.

Sun presentò formalmente Java a una fiera commerciale nel maggio 1995. Normalmente un evento come questo non avrebbe suscitato molta attenzione. Tuttavia, Java generò una curiosità immediata all'interno della comunità affaristica a causa dell'enorme interesse nel World Wide Web. Java è attualmente utilizzato per creare pagine Web dal contenuto dinamico e interattivo, per sviluppare applicazioni aziendali su larga scala, per migliorare la funzionalità dei Web server (i computer che forniscono il contenuto che noi vediamo nei nostri Web browser), per fornire applicazioni per i dispositivi del mercato di massa (come telefoni cellulari, pager e personal digital assistant) e per molti altri scopi.

Nel novembre 1995 stavamo seguendo lo sviluppo di Java da parte di Sun Microsystems e partecipammo a una conferenza su Internet a Boston. Un rappresentante di Sun Microsystems tenne una stimolante presentazione su Java. Man mano che il discorso si sviluppava, ci divenne chiaro che Java avrebbe certamente avuto una parte significativa nello sviluppo di pagine Web interattive e multimediali. Tuttavia ci accorgemmo immediatamente di un potenziale molto maggiore per il linguaggio. Considerammo Java come un linguaggio eccellente per insegnare agli studenti del primo anno del corso di programmazione gli elementi essenziali della grafica, delle immagini, dell'animazione, dell'audio, del video, dei database, delle reti, del multithreading e dell'elaborazione cooperativa.

Oltre alla sua importanza nello sviluppo di applicazioni basate su Internet e Intranet, Java è diventato il linguaggio di scelta per realizzare software per dispositivi in grado di comunicare su una rete (come telefoni cellulari, pager e personal digital assistant). Non rimanete sorpresi il giorno in cui il vostro nuovo stereo e altri dispositivi nella vostra nuova casa saranno collegati assieme per mezzo della tecnologia Java!

1.12 BASIC, Visual Basic, Visual C++, C# e .NET

Il BASIC (Beginner's All-Purpose Symbolic Instruction Code, ovvero, codice di istruzioni simbolico tuttofare per principianti) fu sviluppato a metà degli anni '60 dai professori John Kemeny e Thomas Kurtz del Dartmouth College come un linguaggio per scrivere semplici programmi. Lo scopo principale del BASIC era quello di familiarizzare i principianti con le tecniche di programmazione. Il Visual Basic fu introdotto da Microsoft nel 1991 per semplificare il processo di sviluppo delle applicazioni per Microsoft Windows.

Visual Basic .NET, Visual C++ .NET e C# sono sviluppati per la nuova piattaforma di programmazione .NET della Microsoft. Tutti i tre linguaggi fanno uso della potente libreria di componenti software riusabili del .NET, chiamata Framework Class Library (FCL).

Analogamente a Java, la piattaforma .NET fa in modo che le applicazioni basate sul Web possano essere distribuite a parecchi dispositivi (anche ai telefoni cellulari) e ai computer da tavolo (desktop). Il linguaggio di programmazione C# è stato progettato in modo specifico per la piattaforma .NET come un linguaggio che permetta ai programmatori di migrare facilmente a .NET. Il C++, Java e C# affondano tutti le loro radici nel linguaggio di programmazione C.

1.13 La chiave dell'evoluzione del software; la tecnologia a oggetti

Uno degli autori, HMD, ricorda la grande frustrazione che fu provata negli anni '60 dalle organizzazioni che si occupavano dello sviluppo di software, specialmente quelle che realizzavano progetti su larga scala. Negli anni che precedettero la laurea HMD ebbe il privilegio di lavorare durante i periodi estivi presso uno dei principali venditori di computer nei team di sviluppo di sistemi operativi con timesharing e memoria virtuale. Questa fu un'esperienza significativa per uno studente del college. Tuttavia nel corso dell'estate del 1967 la realtà cambiò quando l'azienda si "disimpegnò" dal realizzare come prodotto commerciale il sistema su cui centinaia di persone avevano lavorato per molti anni. Era difficoltoso mantenere quel programma corretto: il software è "qualcosa di complicato".

I miglioramenti alla tecnologia del software cominciarono a comparire con i benefici della cosiddetta *programmazione strutturata* (e le discipline correlate dell'*analisi e della progettazione dei sistemi strutturati*) che andavano maturando negli anni '70. Tuttavia fu solo quando la tecnologia della programmazione orientata agli oggetti divenne largamente utilizzata negli anni '90, che i programmatori finalmente percepirono di avere gli strumenti necessari per effettuare i passi più significativi nel processo di sviluppo del software.

In effetti, la tecnologia a oggetti risale a metà degli anni '60. Il linguaggio di programmazione C++, sviluppato presso l'AT&T da Bjarne Stroustrup nei primi anni '80, si fonda su due linguaggi, il C e il Simula 67, un linguaggio di programmazione per simulazioni sviluppato in Europa e pubblicato nel 1967. Il C++ ha assimilato le caratteristiche del C e aggiunto le capacità del Simula per creare e manipolare gli oggetti. Né il C né il C++ furono originariamente concepiti per un ampio utilizzo al di fuori dei laboratori di ricerca dell'AT&T. Tuttavia per entrambi si sviluppò rapidamente una base di supporto.

Che cosa sono gli oggetti e perché vengono considerati in modo speciale? Effettivamente la tecnologia a oggetti è uno schema di incapsulazione che permette ai programmatori di creare dei componenti software significativi. Questi ultimi sono di dimensioni ragguardevoli e altamente specializzati su particolari aree applicative. Ci sono oggetti per le date, oggetti per l'ora, oggetti per le buste paga, oggetti per le fatture, oggetti per l'audio, oggetti per il video, oggetti per i file, oggetti per i record e così via. In effetti quasi ogni entità può essere ragionevolmente rappresentata come oggetto.

Viviamo in un mondo di oggetti. È sufficiente guardare intorno a noi. Esistono automobili, aerei, persone, animali, edifici, semafori, ascensori e così via. Prima della comparsa dei linguaggi a oggetti, i linguaggi di programmazione (come il FORTRAN, il Pascal, il BASIC e il C) si focalizzavano sulle azioni (verbi) piuttosto che sulle cose o oggetti (sostantivi). I programmatori che ragionano a oggetti programmano principalmente usando sostantivi. Questo mutamento del paradigma rese difficile scrivere programmi. Ora, con la disponibilità di linguaggi orientati agli oggetti molto diffusi come Java e il C++, i programmatori continuano a ragionare a oggetti e possono programmare in modo orientato agli oggetti. Ciò rappresenta un procedimento più naturale della programmazione procedurale e ha apportato significativi miglioramenti della produttività.

Un problema cruciale della programmazione procedurale consiste nel fatto che le varie unità del programma non riflettono facilmente le entità del mondo reale in modo efficace;

quindi tali unità non sono molto riusabili. Non è un evento raro per i programmatori "partire da zero" in ogni nuovo progetto e dover realizzare del software simile "dal nulla". Ciò rappresenta uno spreco di tempo e denaro dato che le persone "reinventano la ruota" ripetutamente. Grazie alla tecnologia a oggetti, le entità software sviluppate (chiamate *classi*), se progettate in modo appropriato, tendono a essere molto più riusabili nei progetti futuri. Sfruttare librerie di componenti riusabili, come .NET FCL e quelle prodotte da molte altre organizzazioni di sviluppo del software, può ridurre notevolmente lo sforzo richiesto per realizzare certe tipologie di sistemi (paragonato allo sforzo che sarebbe richiesto per reinventare queste funzioni in progetti nuovi).

Alcune organizzazioni riferiscono che il riuso del software non rappresenta in effetti il beneficio principale che esse traggono dalla programmazione orientata agli oggetti. Piuttosto indicano il fatto che la programmazione orientata agli oggetti favorisce la produzione di software che risulta maggiormente comprensibile, meglio organizzato e più facile da mantenere, modificare e correggere. Tale fatto risulta significativo in quanto è stato stimato che una percentuale fino all'80% dei costi del software non è associata agli sforzi iniziali per la sua realizzazione, ma alla continua evoluzione e manutenzione di quest'ultimo durante il suo ciclo di vita.

Qualunque siano i benefici del paradigma a oggetti percepiti, appare chiaro che la programmazione orientata agli oggetti sarà la metodologia di programmazione principale nei prossimi decenni.

1.14 Le basi di un tipico ambiente di sviluppo di un programma C

I sistemi C consistono generalmente di diverse parti: l'ambiente di sviluppo del programma, il linguaggio e la libreria standard del C. La discussione seguente spiegherà il tipico ambiente di sviluppo C, mostrato nella Figura 1.1.

I programmi scritti in C passano tipicamente attraverso 6 fasi, prima di essere eseguiti (Figura 1.1). Queste sono: *editare, preelaborare, compilare, linkare (collegare), caricare* ed *eseguire*. Nonostante questo libro sia un testo generico sul C (scritto in modo indipendente dai dettagli di un particolare sistema operativo), in questa sezione ci concentreremo su un tipico sistema C basato su UNIX. [*Nota*: i programmi in questo libro funzioneranno con piccole modifiche o nessuna variazione del tutto sulla maggior parte dei sistemi C, inclusi i sistemi basati su Microsoft Windows.] Qualora non stiate utilizzando un sistema UNIX, fate riferimento ai manuali del vostro sistema, o chiedete al vostro insegnante di eseguire queste attività nel vostro ambiente.

La prima fase consiste nella scrittura del codice (editing) in un file: questa si esegue con un *programma chiamato editor*. Il `vi` e l'`emacs` sono due editor largamente utilizzati sui sistemi UNIX. I pacchetti software per gli ambienti integrati di sviluppo di programmi C/C++, come Borland C++ Builder e Microsoft Visual Studio, dispongono di editor che sono integrati nell'ambiente di programmazione. Supponiamo che il lettore sappia come editare un programma. Il programmatore scrive un programma in C con l'editor, esegue delle correzioni in caso di necessità, quindi salva il programma su un dispositivo di memoria secondaria, come un disco. Il nome del file del programma C dovrà terminare con l'estensione `.c`.

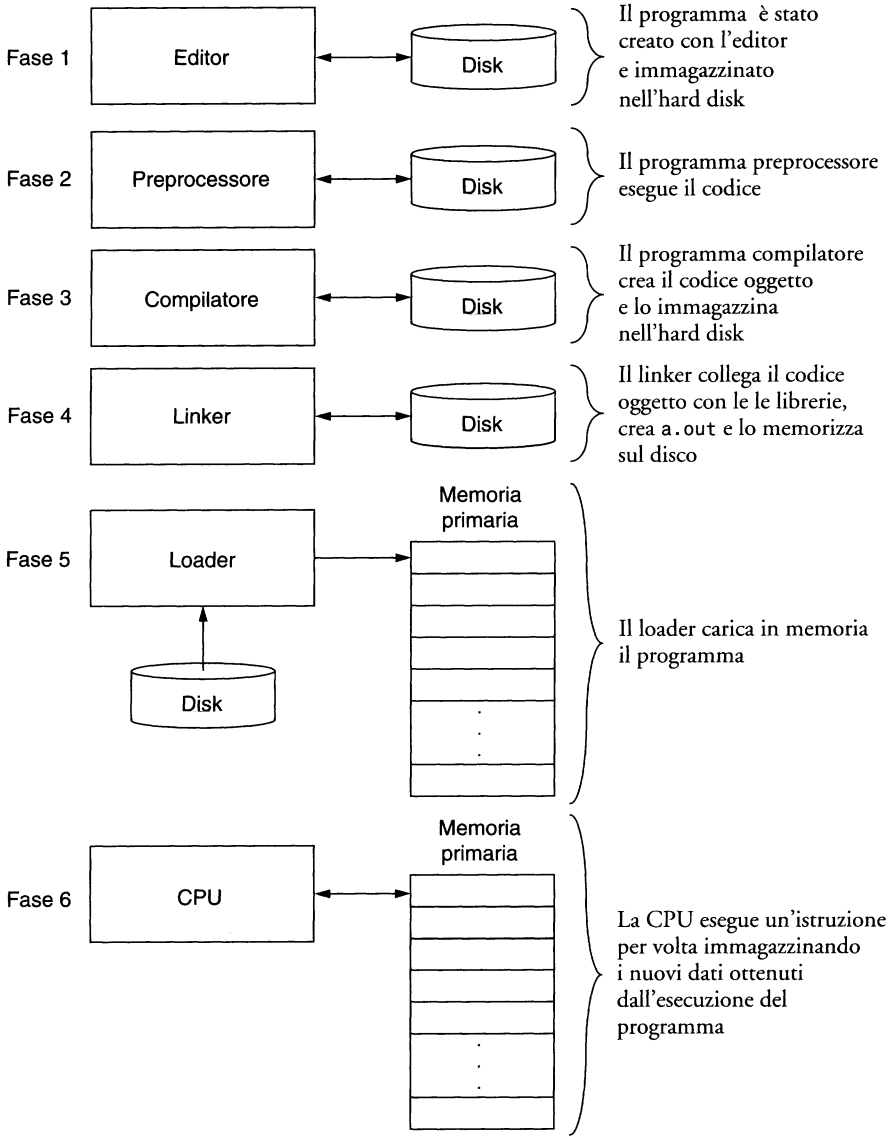


Figura 1.1 Un tipico ambiente C

In seguito, il programmatore immetterà il comando di *compilazione* del programma. Il compilatore tradurrà il programma C nel codice in linguaggio macchina (detto anche *codice oggetto*). In un sistema C, prima che incominci la fase di traduzione del compilatore, sarà eseguito automaticamente il programma *preprocessore*. Il preprocessore del C obbedisce a comandi speciali, chiamati *direttive del preprocessore*, con le quali si indica che sul programma dovranno essere eseguite determinate manipolazioni, prima della compilazione vera e propria. Tali manipolazioni consistono generalmente nella inclusione di altri file in quello da compilare, e nell'esecuzione di vari rimpiazzamenti testuali. Nei primi capitoli, saranno discusse le direttive più comuni del preprocessore; nel Capitolo 13 presenteremo una trattazio-

ne dettagliata delle caratteristiche del preprocessore. Questo sarà invocato automaticamente dal compilatore, prima che il programma sia convertito in linguaggio macchina.

La fase successiva è chiamata *linking (collegamento)*. I programmi scritti in C contengono tipicamente dei riferimenti a funzioni definite altrove, per esempio nelle librerie standard o in quelle private di gruppi di programmatori che lavorano su un particolare progetto. Il codice oggetto prodotto dal compilatore conterrà tipicamente dei "buchi" dovuti a queste parti mancanti. Il *linker* collega il codice oggetto con quello delle funzioni mancanti per produrre una *immagine eseguibile* (senza pezzi mancanti). In un tipico sistema basato su UNIX, il comando per compilare e collegare i pezzi di un programma è *cc*. Per compilare e collegare i pezzi di un programma chiamato `welcome.c` scriverete

```
cc welcome.c
```

al prompt di UNIX e premerete il tasto invio. [*Nota*: i comandi UNIX distinguono fra maiuscole e minuscole; assicuratevi di digitare delle *c* minuscole e che le lettere del nome del file siano appropriatamente minuscole o maiuscole.] Nel caso il programma sia stato compilato e collegato correttamente, sarà prodotto un file chiamato `a.out`. Questo sarà l'immagine eseguibile del nostro programma `welcome.c`.

La fase successiva è chiamata *caricamento*. Prima che possa essere eseguito, un programma dovrà essere caricato nella memoria. Questa operazione sarà eseguita dal *loader (caricatore)* che prenderà l'immagine eseguibile dal disco e la trasferirà nella memoria. Vengono caricati anche i componenti aggiuntivi provenienti dalle librerie condivise che supportano il programma. Finalmente, sotto il controllo della sua CPU, il computer eseguirà il programma, una istruzione per volta. Per caricare ed eseguire il programma in un sistema UNIX, scriveremo `a.out` al prompt di UNIX e premeremo il tasto invio.

I programmi non sempre funzionano al primo tentativo. Ognuna delle fasi precedenti può fallire a causa di vari errori che discuteremo. Per esempio, un programma in esecuzione potrebbe tentare di eseguire una divisione per zero (un'operazione illecita nei computer esattamente come in aritmetica). Ciò causerebbe la stampa a video di un messaggio d'errore da parte del computer. Di conseguenza il programmatore ritornerebbe alla fase di editing, apporterebbe le dovute correzioni e ripeterebbe nuovamente le rimanenti fasi per accertarsi che le correzioni funzionino adeguatamente.

Errore tipico 1.1

Gli errori come la divisione per zero avvengono durante l'esecuzione del programma, quindi sono chiamati errori a run-time o errori a tempo di esecuzione. In generale, la divisione per zero è un errore fatale, ovvero, un errore che provoca la terminazione immediata del programma senza che quest'ultimo possa eseguire con successo il suo compito. Gli errori non fatali consentono ai programmi di andare avanti nell'esecuzione fino al loro completamento, producendo spesso dei risultati non corretti. [Nota: in alcuni sistemi la divisione per zero non è un errore fatale. Si consiglia dunque di consultare la documentazione del proprio sistema.]

La maggior parte dei programmi scritti in C ricevono e/o producono dei dati. Certe funzioni scritte in C prendono il proprio input (i dati in ingresso) dallo *stdin* (il *dispositivo standard per l'input*), che è assegnato normalmente alla tastiera, ma lo `stdin` potrebbe anche essere connesso a un altro dispositivo. L'output (i dati in uscita) è inviato allo *stdout* (il *dispositivo standard per l'output*), che è normalmente lo schermo del computer, ma che potrebbe anche essere

connesso a un altro dispositivo. Quando affermiamo che un programma stampa un risultato, normalmente, intendiamo affermare che quello è visualizzato sullo schermo. I dati possono essere inviati ad altri dispositivi, come i dischi e le stampanti (stampa su carta). C'è anche un *dispositivo standard per l'errore* chiamato *stderr*. Il dispositivo dello `stderr` (normalmente connesso allo schermo) è utilizzato per visualizzare i messaggi di errore. È uso comune dirigere i dati dell'output regolare, ovverosia lo `stdout`, verso un dispositivo diverso dallo schermo, mentre si mantiene lo `stderr` assegnato allo schermo, così che l'utente possa immediatamente essere informato degli errori.

1.15 L'evoluzione dell'hardware

La comunità dei programmatori prospera su un flusso continuo di miglioramenti eclatanti nel campo dell'hardware, del software e delle tecnologie di comunicazione. In generale le persone si aspettano ogni anno di pagare perlomeno un po' di più nuovi prodotti e servizi. L'opposto succede nel campo dei computer e delle comunicazioni, specialmente per quanto riguarda l'hardware che supporta tali tecnologie. Per molti decenni e immutabilmente nell'immediato futuro, i costi dell'hardware sono crollati rapidamente, se non addirittura in modo precipitoso, ogni anno. Si tratta di un fenomeno legato alla tecnologia. Ogni anno oppure ogni due anni le prestazioni dei computer tendono a raddoppiare, mentre i relativi prezzi continuano a calare. La drastica diminuzione del rapporto prezzo/prestazioni dei sistemi computerizzati è guidata dall'aumento in velocità e capacità della memoria in cui i programmi vengono eseguiti dai computer, dall'aumento esponenziale della quantità di memoria secondaria (come lo spazio su disco) in cui vengono memorizzati permanentemente i programmi e i dati e dalla continua crescita della velocità dei processori – velocità alle quali i computer eseguono i loro programmi (ovvero, velocità a cui i computer lavorano).

La stessa crescita è avvenuta nel campo delle comunicazioni, con i costi caduti a picco, specialmente negli ultimi anni dato che l'ingente domanda di larghezza di banda per le comunicazioni ha attirato una fortissima competizione. Non siamo a conoscenza di nessun altro campo in cui la tecnologia si sviluppi in modo così veloce e i costi calino così rapidamente. Quando l'utilizzo dei computer divenne popolare negli anni '60 e '70, ci furono voci sugli enormi miglioramenti della produttività dell'uomo che sarebbero stati apportati dall'elaborazione elettronica e dalle comunicazioni. Tuttavia tali miglioramenti non si concretizzarono. Le organizzazioni spendevano ingenti somme di denaro per i computer e sicuramente li utilizzavano in modo efficace, ma senza realizzare i miglioramenti della produttività che si aspettavano. Furono l'invenzione della tecnologia dei microchip e la sua ampia diffusione nei tardi anni '70 e '80 che prepararono le basi per i miglioramenti della produttività degli anni '90 e del giorno d'oggi.

1.16 La storia di Internet

Nei tardi anni '60, uno degli autori (HMD) era uno studente che si stava specializzando al MIT. La sua ricerca nell'ambito del Progetto Mac del MIT (presso l'attuale Laboratorio di Informatica – sede del World Wide Web Consortium) era finanziata dall'ARPA – Advanced Research Projects Agency of the Department of Defense, l'Agenzia dei Progetti di Ricerca Avanzata del Dipartimento della Difesa. L'ARPA sostenne finanziariamente una conferenza presso l'Università dell'Illinois a Urbana-Champaign a cui parteciparono molti laureati che seguivano corsi di specializzazione finanziati dalla stessa ARPA per incontrarsi e scambiarsi

delle idee. Nel corso della conferenza l'ARPA presentò i progetti per collegare in rete i computer centrali di una dozzina circa di università e istituzioni di ricerca da essa finanziate. Dovevano essere connessi per mezzo di linee di comunicazione operanti all'allora stupefacente velocità di 56 Kb (ovvero, 56.000 bit al secondo), in un'epoca in cui la maggior parte delle persone (delle poche che potevano permetterselo) si collegavano ai computer su linee telefoniche alla velocità di 110 bit al secondo. HMD ricorda chiaramente l'eccitazione di quella conferenza. I ricercatori di Harvard in Massachusetts parlavano di comunicazione attraverso il paese con il "supercomputer" Univac 1108 situato presso l'Università dello Utah per elaborare i calcoli relativi alla loro ricerca nel campo della computer grafica. Molte altre stimolanti possibilità venivano presentate. La ricerca accademica era sul punto di fare un gigantesco balzo in avanti. In breve tempo dopo quella conferenza, l'ARPA procedette a realizzare quella che venne presto chiamata *ARPAnet*, il predecessore dell'odierna Internet.

I fatti si evolsero in modo differente da quanto originalmente pianificato. Nonostante ARPAnet consentisse ai ricercatori di condividere i rispettivi computer l'uno con l'altro, il suo beneficio principale si dimostrò essere la sua capacità di comunicazione facile e veloce per mezzo dello strumento che divenne noto come *posta elettronica (e-mail)*. Ciò è vero ancora oggi su Internet, per mezzo dell'e-mail che facilita comunicazioni di ogni tipo fra centinaia di milioni di persone in tutto il mondo.

Uno degli obiettivi principali dell'ARPA per la rete era di consentire a più utenti di inviare e ricevere informazione allo stesso tempo sugli stessi percorsi di comunicazione (come le linee telefoniche). La rete funzionava con una tecnica chiamata *commutazione di pacchetto* per cui i dati digitali venivano inviati in piccole unità chiamate *pacchetti*. Questi ultimi contenevano i dati, l'informazione sull'indirizzo, i codici di controllo degli errori e l'informazione di sequenziamento. L'informazione sull'indirizzo era utilizzata per instradare i pacchetti verso la loro destinazione. L'informazione di sequenziamento era usata per effettuare il riassemblaggio dei pacchetti (che – a causa della complessità dei meccanismi di instradamento – avrebbero potuto arrivare in modo disordinato) nella loro sequenza originale per poter essere presentati al destinatario. Pacchetti appartenenti a diverse persone si mischiavano nelle linee telefoniche. Questa tecnica di commutazione di pacchetto ridusse enormemente i costi di trasmissione se paragonati a quelli delle linee di comunicazione dedicate.

La rete era progettata per operare senza un controllo centralizzato. Ciò significa che se una parte di essa avesse avuto un guasto, le parti funzionanti rimanenti sarebbero state ancora in grado di instradare i pacchetti dai mittenti ai destinatari su percorsi alternativi.

Il protocollo per comunicare su ARPAnet divenne noto come TCP – Transmission Control Protocol, protocollo di controllo della trasmissione. Il TCP assicurava che i messaggi venissero instradati in modo opportuno dal mittente al destinatario e che arrivassero integri.

Parallelamente alla prima evoluzione di Internet, le organizzazioni in tutto il mondo stavano realizzando le proprie reti sia per la comunicazione entro l'organizzazione che per la comunicazione fra diverse organizzazioni. Comparve un'enorme varietà di hardware e software di rete. Fare in modo che questi ultimi comunicassero fra loro era una sfida. L'ARPA la vinse grazie allo sviluppo di IP – Internetworking Protocol, il protocollo per il collegamento fra reti, creando realmente una "rete delle reti", l'attuale architettura di Internet. L'insieme congiunto di protocolli è ora comunemente chiamato TCP/IP.

Agli inizi l'utilizzo di Internet era limitato ai college e alle istituzioni di ricerca; in seguito le strutture militari divennero un utente importante. Alla fine il governo decise di consentire

l'accesso a Internet per scopi commerciali. Dapprima ci fu risentimento nelle comunità del mondo della ricerca e dei militari – si riteneva che i tempi di risposta sarebbero diventati scadenti non appena "la rete" fosse stata saturata da così tanti utenti.

In realtà avvenne esattamente l'opposto. Le imprese intuirono rapidamente che, tramite un utilizzo efficace di Internet, avrebbero potuto mettere a punto le proprie operazioni e offrire servizi nuovi e migliori ai loro clienti, così iniziarono a investire ingenti somme di denaro per sviluppare e migliorare Internet. Tale fatto generò una feroce competizione fra i fornitori di infrastrutture di comunicazione e fra i fornitori di hardware e software per soddisfare la domanda. Il risultato è che la larghezza di banda (ovvero, la capacità di trasportare informazioni delle linee di comunicazione) su Internet è cresciuta enormemente, mentre i costi sono precipitati. In tutto il mondo i paesi ora comprendono che Internet è di cruciale importanza per la loro prosperità economica e la loro competitività.

1.17 La storia del World Wide Web

Il World Wide Web permette agli utenti di computer di trovare e visualizzare su Internet documenti multimediali (ovvero, documenti composti da testo, grafica, animazioni, audio e/o video) su praticamente qualunque argomento. Nonostante Internet sia stata sviluppata più di trenta anni fa, l'introduzione del World Wide Web è un evento relativamente recente. Nel 1990, Tim Berners-Lee del CERN (il laboratorio europeo per la fisica delle particelle) sviluppò il World Wide Web e molti protocolli di comunicazione che costituiscono la sua spina dorsale.

Internet e il World Wide Web compariranno sicuramente nell'elenco delle più importanti e profonde opere creative dell'umanità. Nel passato la maggior parte delle applicazioni per calcolatori giravano su computer "isolati", ovvero, computer che non erano connessi gli uni agli altri. Le applicazioni odierne possono essere scritte per comunicare fra le centinaia di milioni di computer nel mondo. Internet combina le tecnologie di elaborazione e comunicazione. Rende il nostro lavoro più facile. Fa in modo che l'informazione sia istantaneamente e comodamente accessibile in tutto il mondo. Rende possibile agli individui e alle piccole imprese il fatto di essere visibili a livello mondiale. Sta cambiando il modo di condurre gli affari. La gente può ricercare i prezzi migliori di praticamente ogni prodotto o servizio. Comunità che coltivano interessi particolari possono rimanere in contatto fra loro. I ricercatori possono essere al corrente in modo immediato delle ultime conquiste a livello mondiale.

1.18 Note generali sul C e su questo libro

A volte, i programmatori C esperti sono orgogliosi di essere capaci di creare utilizzi bizzarri, contorti e complicati del linguaggio. Questa è una cattiva abitudine di programmazione. Essa rende i programmi difficili da leggere, maggiormente soggetti a comportamenti strani, più difficili da collaudare e correggere e più difficili da adattare a requisiti mutevoli. Questo libro è adeguato ai programmatori principianti, perciò poniamo l'accento sulla *chiarezza* del programma. La nostra prima "buona abitudine di programmazione" è la seguente.

Buona abitudine 1.1

Scrivete i vostri programmi C in una maniera semplice e diretta. Questa regola è detta a volte KIS ("keep it simple", mantienilo semplice). Non "forzate" il linguaggio, tentando degli utilizzi strani.

Potreste aver sentito dire che il C è un linguaggio portabile e che i programmi scritti in C possono essere eseguiti su molti computer differenti. *La portabilità è un obiettivo sfuggente.* Il documento dello standard ANSI C contiene una lunga lista di problemi riguardanti la portabilità e sull'argomento sono stati scritti libri interi.

Obiettivo portabilità 1.3

Per quanto sia possibile scrivere programmi portabili, ci sono molti problemi tra i tanti compilatori C e i vari computer, che rendono la portabilità un obiettivo difficile da raggiungere. Scrivere semplicemente dei programmi in C non ne garantisce la portabilità. Il programmatore dovrà spesso aver a che fare direttamente con complicate varianti a seconda del computer.

Abbiamo eseguito una attenta verifica del documento dello standard C e abbiamo confrontato con quello la nostra presentazione per completezza e accuratezza. Tuttavia, il C è un linguaggio ricco e ci sono alcune sottigliezze nel linguaggio e certi argomenti avanzati che non abbiamo trattato. Vi suggeriamo di leggere lo stesso documento dello standard C o il libro di Kernighan e Ritchie, nel caso abbiate bisogno di ulteriori dettagli tecnici sul C.

Noi abbiamo preferito limitare la nostra discussione all'ANSI/ISO C. Molte caratteristiche di questa versione del C non sono compatibili con le altre implementazioni del linguaggio, perciò potreste verificare che alcuni dei programmi di questo testo non funzionano sui compilatori C più datati.

Ingegneria del software 1.1

Leggete i manuali della versione del C che state usando. Consultate frequentemente questi manuali per essere sicuri di conoscere la ricca collezione di caratteristiche del C e di usarle in modo corretto.

Ingegneria del software 1.2

Il vostro computer e il vostro compilatore sono dei buoni insegnanti. Qualora non siate sicuri di come funzioni una caratteristica del C, scrivete un programma di esempio con quella caratteristica, compilatelo ed eseguitelo e osservate quello che succede.

Esercizi di autovalutazione

1.1 Riempite gli spazi in ognuna delle seguenti frasi:

a) L'azienda che ha scatenato il fenomeno della elaborazione personale nel mondo è stata la _____.

b) Il computer che ha legittimato l'elaborazione personale nel commercio e nell'industria è stato il _____.

c) I computer elaborano i dati sotto il controllo di insiemi di istruzioni chiamati _____ per computer.

d) Le sei principali unità logiche del computer sono _____, _____, _____, _____, _____ e _____.

e) Il _____ è un caso speciale della multiprogrammazione in cui gli utenti accedono al computer attraverso dispositivi chiamati terminali.

f) Le tre classi di linguaggi discusse in questo capitolo sono _____, _____ e _____.

g) I programmi che traducono in linguaggio macchina quelli scritti in linguaggio di alto livello sono chiamati _____.
h) Il C è largamente conosciuto come il linguaggio di sviluppo del sistema operativo _____.
i) Questo libro presenta la versione del C chiamata _____ C, che è stata standardizzata recentemente dall'Istituto Nazionale Americano per gli Standard.
j) Il linguaggio _____ fu sviluppato da Wirth per insegnare la programmazione strutturata nelle università.
k) Il Dipartimento della Difesa sviluppò il linguaggio Ada con una caratteristica detta _____, che consente ai programmatori di specificare che molte attività possono procedere in parallelo.

1.2 Riempite gli spazi in ognuna delle seguenti frasi riguardanti l'ambiente C.
a) I programmi C sono normalmente immessi in un computer usando un programma _____.
b) In un sistema C, prima che incominci la fase di traduzione, sarà eseguito automaticamente un programma _____.
c) I due tipi più comuni di direttive del preprocessore sono _____ e _____.
d) Il programma _____ combina l'output del compilatore con le varie funzioni di libreria, per produrre una immagine eseguibile.
e) Il programma _____ trasferisce l'immagine eseguibile dal disco alla memoria.
f) Per caricare ed eseguire il programma compilato più recentemente su un sistema UNIX, digitate _____.

Risposte agli esercizi di autovalutazione

1.1 a) Apple. b) Personal Computer della IBM. c) programmi. d) unità di input, unità di output, unità di memoria, unità aritmetica e logica (ALU), unità di elaborazione centrale (CPU), unità di memoria secondaria. e) timesharing. f) linguaggi macchina, linguaggi assembly, linguaggi di alto livello. g) compilatori. h) UNIX. i) ANSI. j) Pascal. k) multitasking.

1.2 a) editor. b) preprocessore. c) includere altri file in quello da compilare, sostituire i simboli speciali con il testo del programma. d) linker. e) loader. f) `a.out`.

Esercizi

1.3 Classificate ognuno dei seguenti elementi come hardware o software:
a) CPU
b) compilatore C
c) ALU
d) preprocessore C
e) unità di input
f) un programma di elaborazione di testi.

1.4 Perché dovreste preferire scrivere un programma in un linguaggio indipendente dalla macchina, invece che in un linguaggio dipendente dalla macchina? Perché un linguaggio dipendente dalla macchina potrebbe essere più appropriato per scrivere certi tipi di programmi?

1.5 I programmi traduttori, come gli assemblatori e i compilatori, convertono i programmi da un linguaggio (detto sorgente) in un altro (detto oggetto). Determinate quali delle seguenti affermazioni sono vere e quali sono false:
a) Un compilatore traduce in linguaggio oggetto i programmi scritti in un linguaggio di alto livello.

b) Un assemblatore converte in programmi in linguaggio macchina quelli scritti in linguaggio sorgente.
c) Un compilatore converte in programmi in linguaggio oggetto quelli scritti in linguaggio sorgente.
d) I linguaggi di alto livello sono generalmente indipendenti dalla macchina.
e) Un programma in linguaggio macchina richiede una traduzione, prima che il programma possa essere eseguito su un computer.

1.6 Riempite gli spazi bianchi delle seguenti frasi:
a) I dispositivi attraverso i quali gli utenti accedono ai sistemi di computer in timesharing sono chiamati di solito _____.
b) Un programma per computer che converte in programmi in linguaggio macchina quelli scritti in linguaggio assembly è chiamato _____.
c) L'unità logica del computer, che riceve dall'esterno le informazioni affinché il calcolatore le possa utilizzare, si chiama _____.
d) Il processo di istruzione del computer per risolvere problemi specifici è chiamato _____.
e) Quale tipo di linguaggio per computer utilizza delle abbreviazioni simili all'inglese per le istruzioni in linguaggio macchina? _____.
f) Quale unità logica del computer invia ai vari dispositivi le informazioni che sono già state elaborate dallo stesso, in modo che queste possano essere utilizzate all'esterno del computer? _____.
g) Il nome generico di un programma, che converte in linguaggio macchina i programmi scritti in un certo linguaggio per computer, è _____.
hi) Quale unità logica del computer conserva le informazioni? _____.
i) Quale unità logica del computer esegue i calcoli? _____.
j) Quale unità logica del computer prende le decisioni logiche? _____.
k) L'abbreviazione comunemente utilizzata per l'unità di controllo del computer è _____.
l) Il livello del linguaggio per computer più conveniente per il programmatore, per scrivere i programmi più velocemente e facilmente, è _____.
m) L'unico linguaggio che un computer può comprendere direttamente è il _____ del computer.
n) Quale unità logica del computer coordina le attività di tutte le altre unità logiche? _____.

1.7 Determinate se ognuna delle seguenti affermazioni sia vera o falsa. Nel caso in cui sia falsa spiegate la vostra risposta.
a) I linguaggi macchina dipendono generalmente dalla macchina.
b) Il timesharing consente realmente a molti utenti di usare simultaneamente un computer.
c) Come gli altri linguaggi di alto livello, il C è generalmente considerato indipendente dalla macchina.

1.8 Discutete il significato di ognuno dei seguenti nomi:
a) `stdin`
b) `stdout`
c) `stderr`

1.9 Perché oggigiorno c'è così tanta attenzione concentrata sulla programmazione orientata agli oggetti, in generale, e sul C++ in particolare?

1.10 Quale linguaggio di programmazione è descritto meglio da ognuna delle seguenti affermazioni?
a) Sviluppato dalla IBM per applicazioni di tipo scientifico e ingegneristico.
b) Sviluppato specificamente per applicazioni commerciali.
c) Sviluppato per insegnare agli studenti la programmazione strutturata.

d) Deve il suo nome al primo programmatore di computer del mondo.
e) Sviluppato per familiarizzare i principianti con le tecniche di programmazione.
f) Sviluppato specificamente per aiutare i programmatori a migrare verso .NET.
g) Conosciuto come il linguaggio di sviluppo di UNIX.
h) Costituito essenzialmente dall'aggiunta al C della programmazione orientata agli oggetti.
i) Deve il suo successo iniziale alla propria predisposizione a creare pagine Web dal contenuto dinamico.

Capitolo 2

Introduzione alla programmazione in C

Obiettivi

- Essere in grado di scrivere semplici programmi per computer in C.
- Essere in grado di usare semplici istruzioni di input e di output.
- Familiarizzare con i tipi di dato fondamentali.
- Comprendere il funzionamento della memoria del computer.
- Essere in grado di usare gli operatori aritmetici.
- Comprendere le regole di priorità degli operatori aritmetici.
- Essere in grado di scrivere semplici istruzioni decisionali.

2.1 Introduzione

Il linguaggio C facilita un approccio strutturato e disciplinato alla progettazione di un programma per computer. In questo capitolo introduciamo la programmazione in C e presentiamo molti esempi che illustrano varie importanti caratteristiche del linguaggio. Ogni esempio è analizzato attentamente, una istruzione per volta. Nel Capitolo 3 e nel Capitolo 4, presentiamo una introduzione alla *programmazione strutturata* in C. In seguito, useremo l'approccio strutturato in tutta la parte rimanente del libro.

2.2 Un semplice programma C: visualizzare una riga di testo

Il C utilizza alcune notazioni che possono apparire strane a chi non ha mai programmato un computer. Cominceremo considerando un semplice programma C. Il nostro primo esempio visualizza una riga di testo. Il programma e la schermata dell'output risultante sono mostrati nella Figura 2.1.

Per quanto questo programma sia semplice, illustra molte caratteristiche importanti del linguaggio C. Consideriamo ora nel dettaglio ogni riga del programma. Le linee 1 e 2

```
/* Fig. 2.1: fig02_01.c
Un primo programma C */
```

incominciano con `/*` e terminano con `*/` per indicare che queste due linee sono un *commento*. I programmatori inseriscono dei commenti per *documentare* i programmi e migliorare la

leggibilità degli stessi. I commenti non provocano l'esecuzione di nessuna azione da parte del computer, quando il programma viene eseguito. I commenti sono ignorati dal compilatore C e non inducono la generazione di alcun codice oggetto in linguaggio macchina. Il commento precedente descrive semplicemente il numero della figura, il nome del file e gli scopi del programma. I commenti aiutano anche le altre persone a leggere e comprendere il vostro programma, ma troppi commenti potrebbero rendere difficile la lettura dello stesso.

```
1  /* Fig. 2.1: fig02_01.c
2     Un primo programma C */
3  #include <stdio.h>
4
5  /* la funzione main è il punto d'inizio dell'esecuzione
       del programma */
6  int main()
7  {
8    printf("Welcome to C!\n");
9
10   return 0; /* indica che il programma è terminato con successo */
11
12 } /* fine della funzione main */
```

```
Welcome to C!
```

Figura 2.1 Un programma che visualizza del testo

Errore tipico 2.1

Dimenticare di terminare un commento con `*/`.

Errore tipico 2.2

Incominciare un commento con i caratteri `*/` o terminarlo con `/*`.

La riga 3

```
#include <stdio.h>
```

è una direttiva per il *preprocessore del C*. Le righe che incominciano con # sono elaborate dal preprocessore, prima che il programma sia compilato. Questa specifica riga indica al preprocessore di includere nel programma il contenuto del *file di intestazione per l'input/output standard* (`stdio.h`). Questo file di intestazione contiene le informazioni usate dal compilatore durante la compilazione delle funzioni per l'input/output standard, come `printf`. Nel Capitolo 5 spiegheremo più dettagliatamente i contenuti dei file di intestazione.

La riga 6

```
int main()
```

è una parte presente in tutti i programmi C. Le parentesi dopo `main` indicano che `main` è un blocco di costruzione del programma, chiamato *funzione*. I programmi C contengono una o

più funzioni, una delle quali deve essere `main`. Ogni programma C comincia eseguendo la funzione `main`.

Buona abitudine 2.1

Ogni funzione dovrebbe essere preceduta da un commento che ne descriva gli scopi.

La *parentesi graffa aperta*, `{`, deve aprire il *corpo* di ogni funzione (riga 7). Una corrispondente *parentesi graffa chiusa* deve chiudere ogni funzione (riga 12). Questa coppia di parentesi graffe e la porzione di programma racchiusa in esse sono anche dette *blocco*. Il blocco è una importante unità dei programmi C.

La riga 8

```
printf("Welcome to C!\n");
```

ordina al computer di eseguire una *azione*: visualizzare sullo schermo la *stringa* di caratteri indicata tra le virgolette. Una stringa è a volte detta *stringa di caratteri*, *messaggio* o *letterale*. L'intera riga, incluso `printf`, i suoi *argomenti* all'interno delle parentesi tonde e il *punto e virgola* (`;`), è detta *istruzione*. Ognuna di queste deve terminare con un punto e virgola (detto anche *terminatore di istruzione*). Nel momento in cui sarà eseguita, l'istruzione `printf` precedente visualizzerà sullo schermo il messaggio `Welcome to C!` Normalmente, i caratteri sono visualizzati nello stesso modo in cui appaiono tra le virgolette della istruzione `printf`. Osservate che i caratteri `\n` non sono stati visualizzati sullo schermo. Il carattere backslash (`\`) è detto *carattere di escape* (fuga). Esso indica alla `printf` che dovrà fare qualcosa fuori dell'ordinario. Nel momento in cui avrà incontrato un backslash in una stringa, il compilatore guarderà il carattere successivo e lo unirà con il backslash per formare una *sequenza di escape*. La sequenza di escape `\n` significa *newline* (nuova riga). Quando una sequenza di newline compare in una stringa stampata da una `printf`, essa provoca il posizionamento del cursore nella posizione iniziale della riga successiva dello schermo. Nella Figura 2.2, sono elencate altre sequenze di escape tipiche.

Le ultime tre sequenze di escape nella Figura 2.2 potrebbero sembrare strane. Dato che il backslash ha un significato speciale in una stringa, ovverosia il compilatore lo riconosce come un carattere di escape, utilizzeremo un doppio backslash (`\\`) per inserire un singolo backslash in una stringa. Anche stampare le virgolette presenta un problema, poiché esse delimitano l'inizio e la fine di una stringa – le virgolette effettivamente non vengono visualizzate. Usando la sequenza di escape `\"` in una stringa da visualizzare con una `printf`, indicheremo che `printf` dovrà stampare le virgolette.

Sequenza di escape	Descrizione
`\n`	Newline (nuova riga). Posiziona il cursore all'inizio della riga successiva.
`\t`	Tabulazione orizzontale. Muove il cursore alla tabulazione successiva.
`\a`	Allarme. Fa suonare il cicalino del sistema.
`\\`	Backslash. Inserisce un carattere backslash in una stringa.
`\"`	Virgolette. Inserisce un carattere virgolette in una stringa.

Figura 2.2 Alcune sequenze di escape tipiche

La riga 10

```
return 0 ; /* indica che il programma è terminato con successo */
```

è inclusa alla fine di ogni funzione `main`. La parola chiave `return` è uno dei molti modi che useremo per *uscire da una funzione*. Quando il comando di ritorno è utilizzato alla fine del main come mostrato in questo caso, il valore `0` indica che il programma è terminato con successo. Nel Capitolo 5 discuteremo in dettaglio le funzioni e le ragioni per cui bisogna includere questo comando risulteranno chiare. Per ora, includeremo questo comando in ogni programma per evitare che il compilatore produca un messaggio di avvertimento su qualche sistema. La parentesi graffa chiusa, `}`, (riga 12) indica che è stata raggiunta la fine del `main`.

Buona abitudine 2.2

Aggiungete un commento alla linea che contiene la parentesi graffa chiusa, `}`*, che termina ogni funzione, incluso il* `main`.

Abbiamo affermato che l'istruzione `printf` spinge il computer a eseguire una *azione*. Nel momento in cui è in esecuzione un programma qualsiasi, il computer svolge una serie di azioni mentre il programma prende delle *decisioni*. Alla fine di questo capitolo, tratteremo delle istruzioni decisionali. Nel Capitolo 3, spiegheremo a fondo questo *modello di azione/decisione* della programmazione.

Errore tipico 2.3

Immettere in un programma il nome della funzione di output `printf` *come print.*

È importante osservare che le funzioni della libreria standard, come `printf` e `scanf`, non fanno parte del linguaggio di programmazione C. Di conseguenza, il compilatore non potrà trovare, per esempio, un errore di battitura in `printf` e `scanf`. Nel momento in cui l'istruzione `printf` sarà compilata, il compilatore fornirà semplicemente uno spazio, nel programma oggetto, per una "chiamata" a quella funzione della libreria. Il compilatore però non sa dove si trovino le funzioni della libreria a differenza del linker che, quando sarà eseguito, individuerà le funzioni della libreria e inserirà nel programma oggetto le chiamate appropriate a queste funzioni. A questo punto il programma oggetto è "completo" e pronto per essere eseguito. Il programma prodotto dal linker, infatti, è spesso chiamato *eseguibile*. Nel caso dovesse esserci un errore di digitazione nel nome della funzione, questo sarà individuato dal linker, perché, per quel nome nel programma C, esso non sarà in grado di trovare una corrispondenza tra i nomi di tutte le funzioni note delle librerie.

Buona abitudine 2.3

L'ultimo carattere stampato da una funzione che esegue qualsiasi visualizzazione dovrebbe essere un newline (`\n`*). Ciò assicura che la funzione lasci il cursore dello schermo nella posizione iniziale di una nuova riga. Le convenzioni di questo genere incoraggiano la riusabilità del software: un obiettivo fondamentale negli ambienti di sviluppo.*

Buona abitudine 2.4

Fate rientrare l'intero corpo di ogni funzione di un livello (tre spazi), all'interno delle parentesi graffe che definiscono il corpo della funzione. Ciò enfatizza la struttura funzionale del programma e aiuta a renderlo più leggibile.

Buona abitudine 2.5

Scegliete una convenzione per la dimensione del rientro che preferite e quindi applicatela uniformemente. Potrebbe essere utilizzato il tasto di tabulazione per creare i rientri, ma tenete presente che i punti di arresto delle tabulazioni potrebbero variare. Raccomandiamo di usare delle tabulazioni da 1/4 di pollice (6,35 mm) oppure di contare manualmente tre spazi per ogni livello di rientro.

La funzione `printf` può visualizzare `Welcome to C!` in molti modi differenti. Per esempio, il programma della Figura 2.3 produce lo stesso output del programma in Figura 2.1. Ciò accade perché ogni `printf` riprende la visualizzazione dal punto in cui si era fermata quella della `printf` precedente. La prima `printf` (riga 8) visualizza `Welcome` seguito da uno spazio, mentre la seconda `printf` (linea 9) incomincia a stampare sulla stessa linea immediatamente dopo lo spazio.

```
1  /* Fig. 2.3: fig02_03.c
2     Visualizzare su una riga con due istruzioni printf */
3  #include <stdio.h>
4
5  /* la funzione main è il punto d'inizio dell'esecuzione
      del programma */
6  int main()
7  {
8     printf("Welcome ");
9     printf("to C!\n");
10
11    return 0; /* indica che il programma è terminato
                   con successo */
12
13 }  /* fine della funzione main */
```

```
Welcome to C!
```

Figura 2.3 Visualizzare su una riga con istruzioni printf distinte

Usando il carattere newline come nella figura 2.4, una singola funzione `printf` potrà visualizzare diverse righe. Ogni volta che incontrerà la sequenza di escape `\n` (newline), la funzione `printf` si posizionerà all'inizio della riga successiva.

```
1  /* Fig. 2.4: fig02_04.c
2     Visualizzare righe multiple con una singola printf */
3   #include <stdio.h>
4
5  /* la funzione main è il punto d'inizio dell'esecuzione
      del programma */
6  int main()
7  {
```

Figura 2.4 Visualizzare righe multiple con una singola printf (continua)

```
 8      printf("Welcome\nto\nC!\n");
 9
10      return 0; /* indica che il programma è terminato con successo */
11
12   } /* fine della funzione main */
```

```
Welcome
to
C!
```

Figura 2.4 Visualizzare righe multiple con una singola printf

2.3 Un altro semplice programma C: sommare due interi

Il nostro prossimo programma utilizza la funzione `scanf` della libreria standard per leggere due interi digitati alla tastiera dall'utente, calcolare la somma dei due valori e visualizzare il risultato usando la funzione `printf`. Il programma e l'output di esempio sono mostrati nella Figura 2.5. [Si noti che nella visualizzazione dell'input/output della Figura 2.5, evidenziamo i numeri inseriti dall'utente.]

```
 1   /* Fig. 2.5: fig02_05.c
 2      Programma di addizione */
 3   #include <stdio.h>
 4
 5   /* la funzione main è il punto d'inizio dell'esecuzione
       del programma */
 6   int main()
 7   {
 8      int integer1; /* primo numero da inserire da parte dell'utente */
 9      int integer2; /* secondo numero da inserire da parte dell'utente */
10      int sum;      /* variabile in cui verrà memorizzata la somma */
11
12      printf("Enter first integer\n");  /* prompt */
13      scanf("%d", &integer1);           /* legge un intero */
14
15      printf("Enter second integer\n"); /* prompt */
16      scanf("%d", &integer2);           /* legge un intero */
17
18      sum = integer1 + integer2;        /* assegnamento della somma */
19
20      printf("Sum is %d\n", sum);       /* visualizza la somma */
21
22      return 0;  /* indica che il programma è terminato con successo */
23
24   }  /* fine della funzione main */
```

Figura 2.5 Un programma per la somma (continua)

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Figura 2.5 Un programma per la somma

I commenti nelle righe 1-2 dichiarano lo scopo del programma. Come abbiamo affermato in precedenza, ogni programma comincia la propria esecuzione con la funzione `main`. La parentesi graffa aperta { (riga 7) marca l'inizio del corpo del `main` mentre la corrispondente parentesi graffa chiusa (riga 24) marca la fine di quest'ultimo.

Le righe 8-10

```
int integer1; /* primo numero da inserire da parte dell'utente */
int integer2; /* secondo numero da inserire da parte dell'utente */
int sum;      /* variabile in cui verrà memorizzata la somma */
```

sono *dichiarazioni*. Le voci `integer1`, `integer2` e `sum` sono i nomi delle *variabili*. Una variabile è una locazione (posizione) della memoria in cui un valore può essere immagazzinato perché possa essere utilizzato da un programma. Questa dichiarazione specifica che le variabili `integer1`, `integer2` e `sum` sono di tipo `int`. Ciò significa che queste variabili conterranno dei valori *interi*, ovverosia, dei numeri interi come 7, –11, 0, 31914 e simili. Tutte le variabili devono essere dichiarate con un nome e un tipo di dato, immediatamente dopo la parentesi graffa aperta che inizia il corpo del `main`, prima che possano essere utilizzate in un programma. In C, ci sono altri tipi di dato oltre a `int`. Si noti che le dichiarazioni precedenti avrebbero potuto essere combinate in una singola dichiarazione come segue:.

```
int integer1, integer2, sum;
```

Un nome di variabile in C è qualsiasi *identificatore* valido. Un identificatore è una serie di caratteri, comprendente lettere, numeri e caratteri di sottolineatura (_), che non comincia con un numero. Un identificatore può avere qualsiasi lunghezza, ma soltanto i primi 31 caratteri devono essere riconosciuti dal compilatore C, in accordo con lo standard dell'ANSI C. Il C è *case sensitive*, ovvero, le lettere maiuscole e quelle minuscole sono differenti in C, perciò `a1` e `A1` sono identificatori distinti.

Errore tipico 2.4

Usare una lettera maiuscola laddove ne dovrebbe essere utilizzata una minuscola (per esempio, scrivendo `Main` *invece di* `main`*).*

Obiettivo portabilità 2.1

Usate identificatori di non più di 31 caratteri. Ciò aiuterà ad assicurare la portabilità e potrà evitare alcuni subdoli errori di programmazione.

Buona abitudine 2.6

Scegliere nomi significativi di variabile aiuterà a rendere il programma auto esplicativo, in altre parole, saranno necessari meno commenti.

Buona abitudine 2.7

La prima lettera di un identificatore usato come un semplice nome di variabile dovrebbe essere una lettera minuscola. Più avanti, in questo libro, assegneremo un significato particolare agli identificatori che incominciano con una lettera maiuscola e a quelli che usano soltanto lettere maiuscole.

Buona abitudine 2.8

I nomi di variabile formati da più parole possono aiutare a rendere il programma più leggibile. Evitate di combinare le diverse parole come in `totalcommissions`. *Separatele piuttosto con caratteri di sottolineatura, come in* `total_commissions` *o, se volete legarle insieme, cominciate ogni parola dopo la prima con una lettera maiuscola, come in* `totalCommissions`.

Le dichiarazioni devono essere sistemate dopo la parentesi graffa aperta di una funzione e prima di *ogni* istruzione eseguibile. Per esempio, nel programma della Figura 2.5, l'inserimento della dichiarazione dopo la prima `printf` avrebbe causato un errore di sintassi. Un *errore di sintassi* è causato quando il compilatore non è in grado di riconoscere una istruzione. Il compilatore normalmente emette un messaggio di errore per aiutare il programmatore a localizzare e correggere l'istruzione errata. Gli errori di sintassi sono violazioni delle regole del linguaggio. Essi sono anche chiamati *errori di compilazione*, o *errori a tempo di compilazione*.

Errore tipico 2.5

Sistemare le dichiarazioni di variabile tra le istruzioni eseguibili provoca degli errori di sintassi.

Buona abitudine 2.9

Separate le dichiarazioni e le istruzioni eseguibili di una funzione con una riga vuota, per enfatizzare il punto in cui finiscono le dichiarazioni e incominciano le istruzioni eseguibili.

La riga 12

```
printf("Enter first integer\n"); /* prompt */
```

visualizza il messaggio `Enter first integer` sullo schermo e si posiziona all'inizio della riga successiva. Questo messaggio è detto *prompt* perché chiede all'utente di eseguire una azione specifica.

L'istruzione

```
scanf("%d", &integer1); /* legge un intero */
```

utilizza `scanf` per ottenere un valore dall'utente. La funzione `scanf` prende i dati in ingresso dallo standard input che è di solito la tastiera. Questa `scanf` ha due argomenti: `"%d"` e `&integer1`. Il primo argomento, la *stringa di controllo del formato*, indica il tipo di dato che dovrà essere immesso dall'utente. La *specifica di conversione* `%d` indica che il dato dovrà essere un intero (la lettera `d` sta per "intero decimale"). Il `%`, in questo contesto, è considerato dalla funzione `scanf` (e come vedremo anche da `printf`) come un carattere speciale che inizia una specifica di conversione. Il secondo argomento della `scanf` incomincia con una E commerciale (`&`), detta in C *operatore di indirizzo*, seguita dal nome della variabile. La E commerciale

(o ampersand), quando è combinata con il nome della variabile, indica alla `scanf` la locazione di memoria in cui è immagazzinata la variabile `integer1`. Il computer quindi immagazzinerà il valore della variabile `integer1` in quella locazione. L'uso di ampersand (`&`) confonde spesso i programmatori principianti o quelli che hanno già programmato in altri linguaggi che non richiedono questa notazione. Per ora, ricordate soltanto di far precedere ogni variabile da un ampersand in tutte le istruzioni `scanf`. Alcune eccezioni a questa regola saranno discusse nel Capitolo 6 e nel Capitolo 7. L'utilizzo di ampersand diventerà chiaro dopo che avremo studiato i puntatori nel Capitolo 7.

Buona abitudine 2.10

Mettete uno spazio dopo ogni virgola (,) per rendere il programma più leggibile.

Nel momento in cui il computer eseguirà la precedente `scanf`, attenderà che l'utente abbia immesso un valore per la variabile `integer1`. L'utente risponderà digitando un intero e premendo il *tasto return* (a volte detto *tasto enter* o *tasto di invio*), per inviare il numero al computer. Questo assegnerà allora il suddetto numero, o *valore*, alla variabile `integer1`. Ogni successivo riferimento a `integer1` nel programma userà tale valore. Le funzioni `printf` e `scanf` facilitano l'interazione tra l'utente e il computer. Dato che questa interazione somiglia a un dialogo, è spesso chiamata *elaborazione interattiva.*

La riga 15

```
printf("Enter second integer\n"); /* prompt */
```

visualizza il messaggio `Enter second integer` sullo schermo e quindi posiziona il cursore all'inizio della riga successiva. Anche questa `printf` chiede all'utente di eseguire una azione.

L'istruzione

```
scanf("%d", &integer2); /* legge un intero */
```

ottiene dall'utente un valore per la variabile `integer2`. L'*istruzione di assegnamento* della riga 18

```
sum = integer1 + integer2; /* assegnamento della somma */
```

calcola la somma delle variabili `integer1` e `integer2` e assegna il risultato alla variabile `sum`, usando l'*operatore di assegnamento* =. L'istruzione è letta come "`sum` *prende* il valore di `integer1 + integer2`". La maggior parte dei calcoli sono eseguiti nelle istruzioni di assegnamento. L'operatore = e quello + sono chiamati *operatori binari* perché ognuno di questi ha due *operandi*. Nel caso dell'operatore +, i due operandi sono `integer1` e `integer2`. Nel caso dell'operatore =, i due operandi sono `sum` e il valore della espressione `integer1 + integer2`.

Buona abitudine 2.11

Inserite degli spazi su entrambi i lati di un operatore binario. Questo mette in risalto l'operatore e rende più leggibile il programma.

Errore tipico 2.6

Il calcolo in una istruzione di assegnamento deve essere a destra dell'operatore =. È un errore di sintassi sistemare un calcolo a sinistra di un operatore di assegnamento.

La riga 20

```
printf("Sum is %d\n", sum); /* visualizza la somma */
```

richiama la funzione `printf` per visualizzare sullo schermo il letterale `Sum is` seguito dal valore numerico della variabile `sum`. Questa `printf` ha due argomenti, `"Sum is %d\n"` e `sum`. Il primo argomento è la stringa di controllo del formato. Questa contiene alcuni caratteri letterali che dovranno essere visualizzati e la specifica di conversione `%d` indicante che sarà visualizzato un intero. Il secondo argomento specifica il valore da visualizzare. Osservate che la specifica di conversione per un intero è la stessa sia in `printf`, sia in `scanf`. Questo accade per la maggior parte dei tipi di dato in C.

I calcoli possono anche essere eseguiti all'interno della istruzione **`printf`**. Avremmo potuto combinare le due precedenti istruzioni in

```
printf("Sum is %d\n", integer1 + integer2);
```

La riga 22

```
return 0; /* indica che il programma è terminato con successo */
```

restituisce il valore `0` all'ambiente del sistema operativo in cui il programma è stato eseguito. Questo indica al sistema operativo che il programma è stato eseguito con successo. Consultate i manuali del vostro specifico sistema operativo, per ottenere informazioni su come comunicare un particolare tipo di fallimento del programma.

La parentesi graffa chiusa, `}`, nella riga 24 indica che è stata raggiunta la fine della funzione `main`.

Errore tipico 2.7

Dimenticare una o entrambe le virgolette che circondano la stringa di controllo del formato in una `printf` *o in una* `scanf`.

Errore tipico 2.8

Dimenticare il `%` *in una specifica di conversione nella stringa di controllo del formato di una* `printf` *o di una* `scanf`.

Errore tipico 2.9

Inserire una sequenza di escape come `\n` *all'esterno della stringa di controllo del formato di una* `printf` *o di una* `scanf`.

Errore tipico 2.10

Dimenticare di includere le espressioni di cui dovranno essere visualizzati i valori, in una `printf` *contenente delle specifiche di conversione.*

Errore tipico 2.11

Non fornire, nella stringa di controllo del formato di una `printf`*, una specifica di conversione, qualora sia necessaria per visualizzare il valore di una espressione.*

Errore tipico 2.12

Inserire, all'interno della stringa di controllo del formato, la virgola che dovrebbe separare la stessa dalla espressione da visualizzare.

Errore tipico 2.13

Dimenticare, in una istruzione `scanf`*, di far precedere da un ampersand una variabile, quando questa dovrebbe, di fatto, essere preceduta da un ampersand.*

Su molti sistemi, il precedente errore rilevato nella fase esecutiva è detto "errore di segmentazione" o "violazione di accesso". Un simile errore occorre quando il programma di un utente tenta di accedere a una parte della memoria del computer per cui il programma stesso non ha diritti di accesso. La causa precisa di questo errore sarà spiegata nel Capitolo 7.

Errore tipico 2.14

Far precedere da un ampersand, in una istruzione `printf`*, una variabile quando, di fatto, questa non dovrebbe essere preceduta da un ampersand.*

2.4 Nozioni sulla memoria

I nomi di variabile come `integer1`, `integer2` e `sum` corrispondono in realtà a *locazioni* (o *posizioni*) nella memoria del computer. Ogni variabile ha un *nome*, un *tipo* e un *valore*.

Nel programma per la somma della Figura 2.5, quando sarà eseguita l'istruzione (riga 13)

```
scanf("%d", &integer1); /* legge un intero */
```

il valore digitato dall'utente sarà immesso nella posizione di memoria alla quale il nome `integer1` è stato assegnato. Supponete che l'utente immetta come valore per `integer1` il numero `45`. Il computer sistemerà il `45` nella locazione `integer1`, come mostrato dalla Figura 2.6.

Ogniqualvolta un valore è sistemato in una posizione di memoria, esso si sostituisce al valore contenuto in precedenza in quella locazione. Dato che questa informazione precedente sarà distrutta, il processo di scrittura in una locazione di memoria è detto *scrittura distruttiva*.

Supponete che l'utente immetta il valore **72** quando l'istruzione (riga 16)

```
scanf("%d", &integer2); /* legge un intero */
```

è eseguita. Questo valore sarà sistemato nella locazione `integer2` e la memoria apparirà come nella Figura 2.7. Osservate che queste posizioni non saranno necessariamente adiacenti in memoria.

Figura 2.6 Una locazione di memoria che mostra il nome e il valore di una variabile

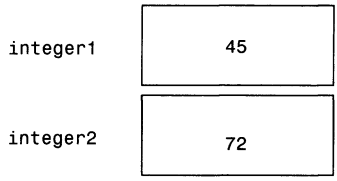


Figura 2.7 Le locazioni di memoria, dopo che entrambe le variabili sono state ricevute in input

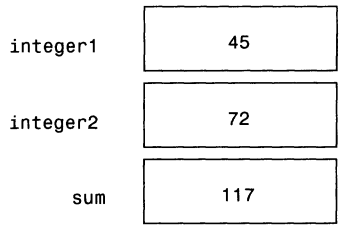


Figura 2.8 Le locazioni di memoria dopo un calcolo

Una volta che il programma avrà ottenuto i valori per `integer1` e `integer2`, esso li sommerà e sistemerà il risultato nella variabile `sum`. L'istruzione (riga 18)

```
sum = integer1 + integer2; /* assegnamento della somma */
```

che esegue l'addizione, comporta anch'essa una scrittura distruttiva. Ciò accadrà quando la somma calcolata da `integer1 + integer2` sarà sistemata nella locazione `sum` (distruggendo il valore che poteva essere già contenuto in `sum`). Dopo che `sum` sarà stata calcolata, la memoria apparirà come nella Figura 2.8. Osservate che i valori di `integer1` e `integer2` appaiono esattamente com'erano prima di essere utilizzati nel calcolo di `sum`. Questi valori sono stati utilizzati, ma non distrutti, quando il computer ha eseguito il calcolo. Di conseguenza, quando un valore è letto da una locazione di memoria, tale processo è detto *lettura non distruttiva.*

2.5 L'aritmetica del C

La maggior parte dei programmi C esegue dei calcoli. Gli *operatori aritmetici* del C sono riassunti nella Figura 2.9. Osservate l'uso di vari simboli speciali non utilizzati in algebra. L'*asterisco* (`*`) indica la moltiplicazione, mentre il *segno di percentuale* (`%`) denota l'operatore *resto* che sarà introdotto in seguito. In algebra, se volessimo moltiplicare *a* per *b,* potremmo semplicemente sistemare questi nomi di variabili, composti di lettere singole, l'uno di fianco all'altro come in *ab.* In C invece, se avessimo scritto così, `ab` sarebbe stato interpretato come un singolo nome (o identificatore), composto di due lettere. Di conseguenza, il C (come gli altri linguaggi di programmazione, in generale) richiede che la moltiplicazione sia esplicitamente denotata, utilizzando l'operatore `*`, come in `a * b`.

Operazioni in C	Operatore aritmetico	Espressione algebrica	Espressione C
Addizione	+	$f + 7$	`f + 7`
Sottrazione	-	$p - c$	`p - c`
Moltiplicazione	*	bm	`b * m`
Divisione	/	x/y o $\frac{x}{y}$ o $x \div y$	`x / y`
Resto	%	$r \bmod s$	`r % s`

Figura 2.9 Gli operatori aritmetici del C

Gli operatori aritmetici sono tutti operatori binari. Per esempio, l'espressione `3 + 7` contiene l'operatore binario + e gli operandi `3` e `7`.

La *divisione tra interi* restituisce un risultato intero. Per esempio, l'espressione `7 / 4` sarà valutata `1`, mentre il valore di `17 / 5` sarà `3`. Il C fornisce l'operatore resto, `%`, che restituisce il resto di una divisione tra interi. Questo è un operatore intero che potrà essere utilizzato soltanto con operandi interi. L'espressione `x % y` restituirà il resto della divisione tra `x` e `y`. Di conseguenza, `7 % 4` restituirà `3`, mentre `17 % 5` produrrà `2`. Discuteremo molte interessanti applicazioni dell'operatore resto.

Errore tipico 2.15

Un tentativo di divisione per zero è di solito indefinito in un computer e generalmente causa un errore fatale, vale a dire, uno di quelli che provocano la terminazione immediata del programma, senza che questo possa completare con successo il proprio lavoro. Gli errori non fatali consentono ai programmi di continuare fino al loro completamento, producendo spesso però dei risultati non corretti.

Le espressioni aritmetiche in C dovranno essere scritte *su una riga* per facilitare l'immissione dei programmi nel computer. Di conseguenza, espressioni come "`a` diviso `b`" dovranno essere scritte come `a/b`, così che tutti gli operatori e gli operandi appaiano su una stessa riga. La notazione algebrica

$$\frac{a}{b}$$

non è generalmente accettabile dai compilatori, sebbene esistano alcuni pacchetti software specializzati, che supportano una notazione più naturale per le espressioni matematiche complesse.

Le parentesi sono utilizzate per raggruppare i termini nelle espressioni C in una maniera molto simile a quella utilizzata nelle espressioni algebriche. Per esempio, per moltiplicare `a` volte la quantità `b + c`, scriveremo:

```
a * (b + c)
```

Il C valuta le espressioni aritmetiche in una sequenza precisa, determinata dalle seguenti *regole di priorità degli operatori*, che sono generalmente le stesse adottate in algebra:

1. In primo luogo saranno valutate le operazioni di moltiplicazione, divisione e resto. Nel caso che un'espressione contenga diverse operazioni di moltiplicazione, divisione e resto, la valutazione procederà da sinistra a destra. La moltiplicazione, la divisione e il resto si trovano allo stesso livello di priorità.
2. In seguito saranno valutate le operazioni di addizione e di sottrazione. Nel caso che una espressione contenga diverse operazioni di addizione e di sottrazione, la valutazione procederà da sinistra a destra. Anche l'addizione e la sottrazione hanno lo stesso livello di priorità, che è minore di quella della moltiplicazione, della divisione e dell'operatore di resto.

Le regole di priorità degli operatori sono direttive che consentono al C di valutare le espressioni nell'ordine corretto. Nel momento in cui affermiamo che la valutazione procede da sinistra a destra, ci stiamo riferendo all'*associatività* degli operatori. Vedremo che alcuni operatori associano da destra a sinistra. La Figura 2.10 riassume le suddette regole di priorità degli operatori.

Operatore/i	Operazione/i	Ordine di valutazione (priorità)
*	Moltiplicazione	Sono valutate per prime. Nel caso che ce ne siano molte, saranno valutate da sinistra a destra.
/	Divisione	
%	Resto	
+	Addizione	Sono valutate in seguito. Nel caso che ce ne siano molte, saranno valutate da sinistra a destra.
-	Sottrazione	

Figura 2.10 Le priorità degli operatori aritmetici

Consideriamo ora diverse espressioni alla luce delle regole di priorità degli operatori. Ogni esempio elenca una espressione algebrica e il suo equivalente in C.

L'esempio seguente calcola la media aritmetica di cinque termini:

Algebra: $m = \dfrac{a+b+c+d+e}{5}$

C:
```
m = (a + b + c + d + e) / 5;
```

Le parentesi sono necessarie per raggruppare gli addendi perché la divisione ha una priorità maggiore della addizione. Dovrà essere divisa per **5** l'intera quantità `(a + b + c + d + e)`. Nel caso le parentesi dovessero essere erroneamente omesse, otterremmo `a + b + c + d + e / 5` che sarebbe valutata in modo scorretto come

$$a+b+c+d+\frac{e}{5}$$

L'esempio seguente è l'equazione di una linea retta:

Algebra: $y = mx + b$

C:
```
y = m * x + b;
```

In questo caso non sono richieste delle parentesi. La moltiplicazione sarà valutata per prima poiché essa ha una priorità maggiore della addizione.

L'esempio successivo contiene le operazioni di resto (`%`), moltiplicazione, divisione, addizione, sottrazione e assegnamento:

Algebra: $z = pr\%q + w/x - y$

C:
```
z = p * r % q + w / x - y;
    ⑥ ①  ②  ④  ③  ⑤
```

I numeri cerchiati sotto l'istruzione indicano l'ordine in cui il C valuterà gli operatori. La moltiplicazione, il resto e la divisione saranno valutati per primi in ordine da sinistra a destra (ovverosia associano da sinistra a destra), poiché hanno una priorità maggiore della addizione e della sottrazione. In seguito saranno valutate l'addizione e la sottrazione. Anche queste saranno valutate da sinistra a destra.

Non tutte le espressioni con diverse coppie di parentesi contengono delle parentesi nidificate. L'espressione

```
a * (b + c) + c * (d + e)
```

non contiene parentesi nidificate. Diremo invece che le parentesi sono "allo stesso livello".

Per sviluppare una miglior comprensione delle regole di priorità degli operatori, vediamo come il C valuta un polinomio di secondo grado.

```
y = a * x * x + b * x + c;
  ⑥ ① ② ④ ③ ⑤
```

I numeri cerchiati sotto l'istruzione indicano l'ordine in cui il C eseguirà le operazioni. Non esiste nessun operatore aritmetico in C per l'elevamento a potenza, perciò abbiamo rappresentato `x2` come `x * x`. La libreria standard del C include la funzione `pow` ("power", potenza) per eseguire l'elevamento a potenza. A causa di alcuni insidiosi problemi correlati ai tipi di dato richiesti da `pow`, rimandiamo la spiegazione dettagliata di `pow` fino al Capitolo 4.

Supponete che `a = 2`, `b = 3`, `c = 7` e `x = 5`. La Figura 2.11 mostra come sarà valutato il precedente polinomio di secondo grado.

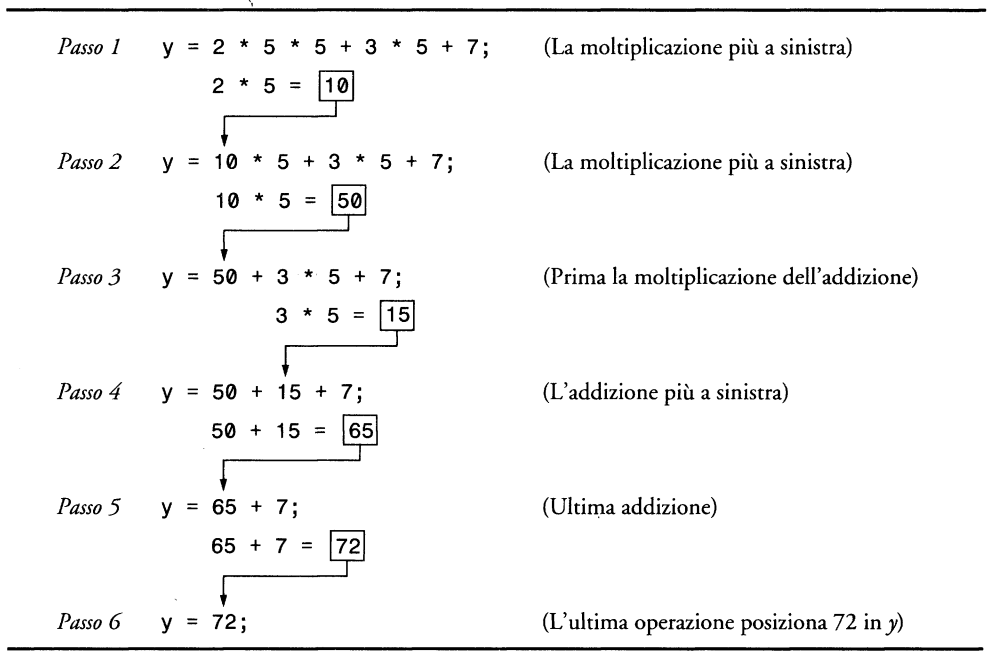


Figura 2.11 Valutazione di un polinomio di secondo grado

2.6 Prendere delle decisioni: gli operatori di uguaglianza e relazionali

Le istruzioni eseguibili del C compiono delle *azioni* (come dei calcoli o l'input e l'output dei dati), oppure prendono delle *decisioni* (vedremo presto molti esempi di questo tipo). In un programma, per esempio, potremmo prendere una decisione per determinare se il voto di una persona in un esame è maggiore o uguale a 60 e, in tal caso, visualizzare il messaggio "Congratulazioni! Avete superato l'esame". Questa sezione introduce una semplice versione

dell'istruzione `if` del C, che consente a un programma di prendere delle decisioni basate sulla verità o sulla falsità di espressioni chiamate per l'appunto *condizioni*. Nel caso in cui la condizione sia stata soddisfatta, ovverosia nel caso in cui risulti *vera*, verrà eseguita l'istruzione presente all'interno del corpo dell'istruzione `if`. Nel caso in cui la condizione non sia soddisfatta ovverosia nel caso in cui risulti *falsa*, l'istruzione presente all'interno del corpo non verrà eseguita. Dopo il completamento del comando `if`, indipendentemente dal fatto che l'istruzione del corpo sia eseguita o no, l'esecuzione procederà con l'istruzione successiva al comando `if`.

Le condizioni nelle istruzioni `if` sono formate usando gli *operatori di uguaglianza* e gli *operatori relazionali* riassunti nella Figura 2.12. Gli operatori relazionali hanno tutti lo stesso livello di priorità e associano da sinistra a destra. Gli operatori di uguaglianza hanno un livello di priorità più basso degli operatori relazionali e associano da sinistra a destra. [*Nota*: in C, una condizione può essere in realtà una espressione che generi un valore uguale (falso) o diverso (vero) da zero. Vedremo molte applicazioni di ciò nel corso del libro].

Operatori algebrici di uguaglianza o relazionali standard	Operatori di uguaglianza o relazionali in C	Esempio di condizione C	Significato della condizione C
Operatori di uguaglianza			
=	==	`x == y`	x è uguale a y
≠	!=	`x != y`	x non è uguale a y
Operatori relazionali			
>	>	`x > y`	x è maggiore di y
<	<	`x < y`	x è minore di y
≥	>=	`x >= y`	x è maggiore o uguale a y
≤	<=	`x <= y`	x è minore o uguale a y

Figura 2.12 Operatori di uguaglianza e relazionali

Errore tipico 2.16

Sarà generato un errore di sintassi se si separeranno con degli spazi i due simboli che compongono ognuno degli operatori ==, != , >= *e* <=.

Errore tipico 2.17

Sarà generato un errore di sintassi se si invertiranno i due simboli che compongono ognuno degli operatori != , >= *e* <=, *rispettivamente in* =!, => *e* =<.

Errore tipico 2.18

Confondere l'operatore di uguaglianza == *con quello di assegnamento* =.

Per evitare questa confusione, l'operatore di uguaglianza dovrebbe essere letto come "doppio uguale" mentre l'operatore di assegnamento dovrebbe essere letto come "prende". Confondere questi operatori, come vedremo presto, non provocherà necessariamente un errore di sintassi facilmente riconoscibile, ma potrà causare degli errori logici molto subdoli.

Errore tipico 2.19

Inserire un punto e virgola subito dopo la parentesi destra della condizione, in un'istruzione `if`.

In Figura 2.13 si utilizzano sei istruzioni `if` per confrontare due numeri immessi dall'utente. Qualora la condizione in qualcuna di queste istruzioni `if` dovesse essere soddisfatta, sarebbe eseguita la funzione `printf` associata a quell'`if`. Nella figura sono mostrati il programma e tre output di esempio.

```
1  /* Fig. 2.13: fig02_13.c
2     Usare le istruzioni if, gli operatori relazionali
3     e quelli di uguaglianza */
4  #include <stdio.h>
5
6  /* la funzione main è il punto d'inizio dell'esecuzione
      del programma */
7  int main()
8  {
9     int num1; /* primo numero da inserire da parte dell'utente */
10    int num2; /* secondo numero da inserire da parte dell'utente */
11
12    printf(„Enter two integers, and I will tell you\n");
13    printf("the relationships they satisfy: ");
14
15    scanf("%d%d", &num1, &num2);   /* legge due interi */
16
17    if (num1 == num2) {
18       printf("%d is equal to %d\n", num1, num2);
19    } /* fine dell'istruzione if */
20
21    if (num1 != num2) {
22       printf("%d is not equal to %d\n", num1, num2);
23    } /* fine dell'istruzione if */
24
25    if (num1 < num2) {
26       printf("%d is less than %d\n", num1, num2);
27    } /* fine dell'istruzione if */
28
29    if (num1 > num2) {
30       printf("%d is greater than %d\n", num1, num2);
31    } /* fine dell'istruzione if */
32
33    if (num1 <= num2) {
34       printf("%d is less than or equal to %d\n", num1, num2);
35    } /* fine dell'istruzione if */
36
37    if (num1 >= num2) {
38       printf(„%d is greater than or equal to %d\n", num1, num2);
```

Figura 2.13 Usare gli operatori di uguaglianza e relazionali (continua)

```
39      } /* fine dell'istruzione if */
40
41      return 0;   /* indica che il programma è terminato con successo */
42
43   } /* fine della funzione main */
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Figura 2.13 Usare gli operatori di uguaglianza e relazionali

Osservate che il programma in Figura 2.13 utilizza `scanf` (riga 15) per prendere in input i due numeri. Ogni specifica di conversione ha un argomento corrispondente in cui dovrà essere immagazzinato un valore. Il primo `%d` convertirà un valore che sarà immagazzinato nella variabile `num1`, mentre il secondo convertirà un valore che sarà immagazzinato nella variabile `num2`. Far rientrare il corpo di ogni istruzione `if` e inserire delle righe vuote sopra e sotto ognuna di queste, aumenterà la leggibilità del programma. Notate anche che ogni `if` della Figura 2.13 ha una singola istruzione nel suo corpo. Nel Capitolo 3, mostreremo come specificare delle `if` con corpi composti da più istruzioni.

Buona abitudine 2.12

Fare rientrare la/e istruzione/i nel corpo di un'istruzione `if`.

Buona abitudine 2.13

Inserire, per la leggibilità, una riga vuota prima e dopo ogni istruzione `if` *in un programma.*

Buona abitudine 2.14

Nonostante sia permesso, non ci dovrebbe essere più di una istruzione per riga in un programma.

Errore tipico 2.20

Inserire delle virgole (quando non sono richieste) tra le specifiche di conversione nella stringa di controllo del formato in una istruzione `scanf`.

Il commento (linee 1-3) nella Figura 2.13 è spezzato su tre righe. Nei programmi C, i caratteri di *spazio bianco* come le tabulazioni, i newline e gli spazi, sono normalmente ignorati. Di conseguenza, le istruzioni e i commenti potranno essere spezzati su più righe. Non è in ogni caso corretto spezzare gli identificatori.

Buona abitudine 2.15

Una istruzione lunga potrà essere distribuita su molte righe. Nel caso che una istruzione possa essere spezzata su più righe, scegliete dei punti di interruzione che abbiano un senso (per esempio dopo una virgola in un elenco separato da virgole). Nel caso che una istruzione sia stata spezzata su due o più righe, fate rientrare tutte quelle successive alla prima.

La tabella nella Figura 2.14 mostra le priorità degli operatori introdotti in questo capitolo. Gli operatori sono mostrati dall'alto in basso, in ordine decrescente di priorità. Osservate che anche il segno di uguale è un operatore. Tutti questi operatori, a eccezione di quello di assegnamento =, associano da sinistra a destra. L'operatore di assegnamento (=) associa da destra a sinistra.

Buona abitudine 2.16

Ogni volta che scrivete delle espressioni contenenti molti operatori, fate riferimento alla relativa tabella delle priorità. Assicuratevi che gli operatori nella espressione siano eseguiti nell'ordine appropriato. Nel caso non siate sicuri di un ordine di valutazione in una espressione complessa, usate le parentesi per raggruppare le espressioni. Assicuratevi di osservare che alcuni operatori del C, come quello di assegnamento (=)*, associano da destra a sinistra, invece che da sinistra a destra.*

Alcune delle parole che abbiamo usato nei programmi C di questo capitolo, in particolare `int`, `return` e `if`, sono keyword (parole chiave) ovverosia parole riservate del linguaggio. Le parole chiave del C sono mostrate nella Figura 2.15. Queste hanno un significato speciale per il compilatore C, perciò il programmatore dovrà stare attento a non utilizzarle come identificatori, per esempio, nei nomi di variabile. In questo libro, discuteremo di tutte le parole chiave.

In questo capitolo, abbiamo introdotto molte caratteristiche importanti del linguaggio di programmazione C, inclusa la visualizzazione dei dati sullo schermo, l'immissione degli stessi da parte dell'utente, l'esecuzione di calcoli e le istruzioni decisionali. Nel prossimo capitolo, ci baseremo su queste tecniche mentre introdurremo la *programmazione strutturata*. Acquisirete maggiore dimestichezza con la tecnica di applicare dei rientri al codice. Studieremo come specificare l'ordine in cui le istruzioni saranno eseguite: il cosiddetto *flusso di controllo*.

Operatori	Associatività
`( )`	da sinistra a destra
`* / %`	da sinistra a destra
`+ -`	da sinistra a destra
`< <= > >=`	da sinistra a destra
`== !=`	da sinistra a destra
`=`	da destra a sinistra

Figura 2.14 Priorità e associatività degli operatori discussi finora

Parole chiave

`auto`	`break`	`case`	`char`
`const`	`continue`	`default`	`do`
`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`
`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`
`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`

Figura 2.15 Le parole chiave riservate del C

Esercizi di autovalutazione

2.1 Riempite gli spazi in ognuna delle seguenti righe.

a) Ogni programma C incomincia la propria esecuzione con la funzione _____.
b) La _____ inizia il corpo di ogni funzione mentre la _____ termina il corpo di ogni funzione.
c) Ogni istruzione termina con un _____.
d) La funzione _____ della libreria standard visualizza le informazioni sullo schermo.
e) La sequenza d'escape `\n` rappresenta il carattere _____ che muove il cursore nella posizione iniziale della riga successiva sullo schermo.
f) La funzione _____ della libreria standard è utilizzata per leggere i dati dalla tastiera.
g) La specifica di conversione _____ è usata in una stringa di controllo del formato di una funzione `scanf`, per indicare che sarà immesso un intero, mentre in una stringa di controllo del formato di una funzione `printf`, per indicare che sarà visualizzato un intero.
h) Ogniqualvolta un nuovo valore sarà sistemato in una locazione di memoria, esso si sostituirà al valore contenuto in precedenza in quella locazione. Questo processo è noto come _____.
i) Quando un valore sarà letto da una locazione di memoria, il valore in quella locazione sarà preservato; questo processo è detto _____.
j) L'istruzione _____ è utilizzata per prendere delle decisioni.

2.2 Stabilite quali delle seguenti affermazioni sono vere e quali false. Nel caso siano false, spiegatene il motivo.

a) Quando sarà invocata la funzione `printf`, questa comincerà a visualizzare sempre dall'inizio di una nuova riga.
b) I commenti inducono il computer a visualizzare sullo schermo il testo racchiuso tra `/*` e `*/` durante l'esecuzione del programma.
c) La sequenza di escape `\n`, quando è usata in una stringa di controllo del formato di una funzione `printf`, induce il cursore a posizionarsi all'inizio della riga successiva sullo schermo.
d) Tutte le variabili devono essere dichiarate prima di essere utilizzate.
e) A tutte le variabili dovrà essere assegnato un tipo, quando saranno dichiarate.
f) Il C considera identiche le variabili `number` e `NuMbEr`.
g) Le dichiarazioni possono apparire in qualsiasi parte del corpo di una funzione.
h) Tutti gli argomenti successivi alla stringa di controllo del formato in una funzione `printf` devono essere preceduti da un ampersand `(&)`.

i) L'operatore modulo (`%`) può essere utilizzato soltanto con degli operandi interi.
j) Gli operatori aritmetici `*`, `/`, `%`, `+` e `-` hanno tutti lo stesso livello di priorità.
k) I seguenti nomi di variabili sono identici in tutti i sistemi ANSI C.

```
thisisasuperduperlongname1234567
thisisasuperduperlongname1234568
```

l) Un programma che visualizzi tre righe di output dovrà contenere tre istruzioni `printf`.

2.3 Scrivete una singola istruzione C per eseguire ognuno dei seguenti compiti:
a) Dichiarate le variabili `c`, `thisVariable`, `q76354` e `number` di tipo `int`.
b) Richiedete all'utente di immettere un intero. Terminate il vostro messaggio di richiesta con un due punti (`:`) seguito da uno spazio e lasciate il cursore posizionato dopo lo spazio.
c) Leggete in input un intero dalla tastiera e immagazzinate il valore immesso nella variabile intera `a`.
d) Se `number` non è uguale a `7`, visualizzate `"The variable number is not equal to 7"`.
e) Visualizzate il messaggio `"This is a C program."` su una riga.
f) Visualizzate il messaggio `"This is a C program."` su due righe, dove la prima termina con `C`.
g) Visualizzate il messaggio `"This is a C program."` con ogni parola su una riga separata.
h) Visualizzate il messaggio `"This is a C program."` con ogni parola separata da una tabulazione.

2.4 Scrivete una istruzione (o un commento) per eseguire ognuno dei seguenti compiti:
a) Indicate che un programma calcolerà il prodotto di tre interi.
b) Dichiarate le variabili `x`, `y`, `z` e `result` di tipo `int`.
c) Richiedete all'utente di immettere tre interi.
d) Leggete in input tre interi dalla tastiera e immagazzinateli nelle variabili `x`, `y` e `z`.
e) Calcolate il prodotto dei tre interi contenuti nelle variabili `x`, `y` e `z`, e assegnate il risultato alla variabile `result`.
f) Visualizzate `"The product is"` seguito dal valore della variabile `result`.

2.5 Usando le istruzioni che avete scritto nell'Esercizio 2.4, scrivete un programma completo che calcoli il prodotto di tre interi.

2.6 Identificate e correggete gli errori in ognuna delle seguenti istruzioni:
a)
```
printf("The value is %d\n", &number);
```
b)
```
scanf("%d%d", &number1, number2);
```
c)
```
if (c < 7);
   printf("C is less than 7\n");
```
d)
```
if (c => 7)
   printf("C is equal to or less than 7\n");
```

Risposte agli esercizi di autovalutazione

2.1 a) main. b) parentesi graffa aperta (`{`). c) parentesi graffa chiusa (`}`). c) punto e virgola. d) `printf`. e) newline. f) `scanf`. g) `%d`. h) scrittura distruttiva. i) lettura non distruttiva. j) `if`.

2.2 a) Falso. La funzione `printf` incomincerà sempre a visualizzare laddove sarà stato sistemato il cursore, e questo potrà essere ovunque su una riga dello schermo.
b) Falso. I commenti non causano l'esecuzione di nessuna azione durante l'esecuzione del programma. Essi sono usati per documentare i programmi e migliorare la loro leggibilità.
c) Vero.
d) Vero.
e) Vero.

f) Falso. Il C distingue fra maiuscole e minuscole, perciò queste variabili sono diverse.
g) Falso. Le dichiarazioni dovranno apparire dopo la parentesi graffa aperta del corpo di una funzione e prima di ogni istruzione eseguibile.
h) Falso. Gli argomenti in una funzione `printf` di norma non devono essere preceduti da un ampersand. Gli argomenti successivi alla stringa per il controllo del formato di una funzione `scanf` dovranno essere preceduti di norma da un ampersand. Discuteremo delle eccezioni nei Capitoli 6 e 7.
i) Vero.
j) Falso. Gli operatori `*`, `/` e `%` hanno lo stesso livello di priorità, mentre gli operatori `+` e `-` sono a un livello di priorità più basso.
k) Falso. Alcuni sistemi possono distinguere degli identificatori con più di 31 caratteri.
l) Falso. Una istruzione `printf` con molte sequenze di escape `\n` può visualizzare tante righe.

2.3 a)
```
int c, thisVariable, q76354, number;
```
b)
```
printf("Enter an integer: ");
```
c)
```
scanf(„%d", &a);
```
d)
```
if (number != 7)
   printf("The variable number is not equal to 7.\n");
```
e)
```
printf("This is a C program.\n");
```
f)
```
printf("This is a C\nprogram.\n");
```
g)
```
printf("This\nis\na\nC\nprogram.\n");
```
h)
```
printf("This\tis\ta\tC\tprogram.\n");
```

2.4 a)
```
/* Calcola il prodotto di tre interi */
```
b)
```
int x, y, z, result;
```
c)
```
printf("Enter three integers: ");
```
d)
```
scanf("%d%d%d", &x, &y, &z);
```
e)
```
result = x * y * z;
```
f)
```
printf("The product is %d\n", result);
```

2.5 Si veda il listato sottostante.

```
1  /* Calcola il prodotto di tre interi */
2  #include <stdio.h>
3
4  int main()
5  {
6     int x, y, z, result; /* dichiarazione delle variabili */
7
8     printf("Enter three integers: ");  /*prompt */
9     scanf("%d%d%d", &x, &y, &z); /* legge tre interi */
10    result = x * y * z; /* moltiplica I valori */
11    printf("The product is %d\n", result); /* visualizza il risultato */
12
13    return 0;
14 }
```

2.6 a) Errore: `&number`. Correzione: eliminate il carattere `&`. Più tardi nel testo discuteremo delle eccezioni a questa regola.
b) Errore: `number2` non ha un ampersand. Correzione: `number2` deve essere `&number2`. Più avanti nel libro discuteremo delle eccezioni a questa regola.
c) Errore: il punto e virgola dopo la parentesi destra della condizione dell'istruzione `if`. Correzione: rimuovete il punto e virgola dopo la parentesi destra. [*Nota*: il risultato di questo errore sarà che l'istruzione `printf` sarà eseguita indipendentemente dal fatto che la condizione della istruzione `if` sia vera o no. Il punto e virgola dopo la parentesi destra sarà considerato alla stessa stregua di una istruzione vuota: una istruzione che non fa niente.]

d) Errore: l'operatore relazionale =>, dovrà essere cambiato in >= (maggiore o uguale a).

Esercizi

2.7 Identificate e correggete gli errori in ognuna delle seguenti istruzioni (*Nota*: potrebbe esserci più di un errore per istruzione):

a)
```
scanf("d", value);
```
b)
```
printf("The product of %d and %d is %d"\n, x, y);
```
c)
```
firstNumber + secondNumber = sumOfNumbers
```
d)
```
if (number => largest)
   largest == number;
```
e)
```
*/ Program to determine the largest of three integers /*
```
f)
```
Scanf("%d", anInteger);
```
g)
```
printf("Remainder of %d divided by %d is\n", x, y, x % y);
```
h)
```
if (x = y);
   printf(%d is equal to %d\n", x, y);
```
i)
```
print("The sum is %d\n," x + y);
```
j)
```
Printf("The value you entered is: %d\n, &value);
```

2.8 Riempite gli spazi in ognuna delle seguenti righe:

a) I _____ sono utilizzati per documentare un programma e migliorarne la leggibilità.
b) La funzione utilizzata per visualizzare le informazioni sullo schermo è _____.
c) Una istruzione C che prenda una decisione è una _____.
d) I calcoli sono normalmente eseguiti dalle istruzioni di _____.
e) La funzione _____ prende in input dalla tastiera dei valori.

2.9 Scrivete una singola istruzione C o una riga che esegue ognuno dei seguenti compiti:

a) Visualizzate il messaggio `"Enter two numbers."`
b) Assegnate alla variabile `a` il prodotto di `b` e `c`.
c) Indicate che un programma esegue un esempio di calcolo dello stipendio (ovverosia, utilizzate un testo che aiuti a documentare un programma).
d) Prendete in input dalla tastiera tre valori interi e memorizzateli nelle variabili `a`, `b` e `c`.

2.10 Stabilite quale delle seguenti affermazioni è vera e quale è falsa. Nel caso un'affermazione sia falsa motivate la vostra risposta.

a) Gli operatori del C sono valutati da sinistra a destra.
b) Quelli che seguono sono tutti nomi di variabile validi: `_under_bar_`, `m928134`, `t5`, `j 7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
c) L'istruzione `printf("a = 5;");` è un tipico esempio di istruzione di assegnamento.
d) Una espressione aritmetica valida, che non contiene parentesi, sarà valutata da sinistra a destra.
e) Quelli che seguono sono tutti nomi di variabile non validi: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Riempite gli spazi bianchi in ognuna delle seguenti righe:

a) Quali operazioni aritmetiche hanno lo stesso livello di priorità della moltiplicazione? _____.
b) Quando le parentesi sono nidificate, quale gruppo di parentesi sarà valutato per primo in una espressione aritmetica? _____.
c) Una locazione della memoria di un computer che può contenere valori differenti, in vari momenti della esecuzione di un programma, è una _____.

2.12 Che cosa sarà visualizzato (se lo sarà), quando ognuna delle seguenti istruzioni verrà eseguita? Nel caso in cui non venga visualizzato niente, rispondete "niente". Assumete che `x = 2` e `y = 3`.

a)
```
printf("%d", x);
```

b) `printf("%d", x + x);`
c) `printf("x=");`
d) `printf("x=%d", x);`
e) `printf("%d = %d", x + y, y + x);`
f) `z = x + y;`
g) `scanf("%d%d", &x, &y);`
h) `/* printf("x + y = %d", x + y); */`
i) `printf("\n");`

2.13 Quale delle seguenti istruzioni C, se ce n'è una, contiene delle variabili coinvolte in una lettura distruttiva?

a) `scanf("%d%d%d%d%d", &b, &c, &d, &e, &f);`
b) `p = i + j + k + 7;`
c) `printf("Destructive read-in");`
d) `printf("a = 5");`

2.14 Data l'equazione `y = ax3 + 7`, quale delle seguenti istruzioni C, se ce n'è una, è quella corretta per questa equazione?

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * (x + 7);`
c) `y = (a * x) * x * (x + 7);`
d) `y = (a * x) * x * x + 7;`
e) `y = a * (x * x * x) + 7;`
f) `y = a * x * (x * x + 7);`

2.15 Indicate l'ordine di valutazione degli operatori, in ognuna delle seguenti istruzioni C, e mostrate il valore di `x` dopo che ogni istruzione sarà stata eseguita.

a) `x = 7 + 3 * 6 / 2 - 1;`
b) `x = 2 % 2 + 2 * 2 - 2 / 2;`
c) `x = (3 * 9 * (3 + (9 * 3 / (3))));`

2.16 Scrivete un programma che chieda all'utente di immettere due numeri, ottenga i numeri dall'utente e visualizzi la loro somma, prodotto, differenza, quoziente e resto.

2.17 Scrivete un programma che visualizzi i numeri da 1 a 4 sulla stessa riga. Scrivete il programma utilizzando i seguenti metodi:

a) Usando una istruzione `printf` senza specifiche di conversione.
b) Usando una istruzione `printf` con quattro specifiche di conversione.
c) Usando quattro istruzioni `printf`.

2.18 Scrivete un programma che chieda all'utente di immettere due interi, ottenga i numeri e visualizzi quello maggiore seguito dalle parole `"is larger."`. Nel caso che i numeri siano uguali, stampate il messaggio `"These numbers are equal."`. Usate soltanto la forma a selezione singola della istruzione `if` che avete appreso in questo capitolo.

2.19 Scrivete un programma che prenda in input dalla tastiera tre diversi interi e quindi visualizzi la somma, la media, il prodotto, il minore e il maggiore di questi numeri. Usate soltanto la forma a selezione singola della istruzione `if` che avete appreso in questo capitolo. Lo schermo di dialogo dovrà apparire come il seguente:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20 Scrivete un programma che legga il raggio di un cerchio e visualizzi il diametro, la circonferenza e l'area dello stesso. Usate il valore costante 3,14159 per π. Eseguite ognuno di questi calcoli all'interno della/e istruzione/i `printf` e usate la specifica di conversione `%f`. [*Nota*: in questo capitolo abbiamo trattato soltanto costanti e variabili intere. Nel Capitolo 3 discuteremo i numeri in virgola mobile, ovverosia quei valori che possono contenere dei decimali.]

2.21 Scrivete un programma che visualizzi una scatola, un ovale, una freccia e un diamante come i seguenti:

```
*********          ***              *                *
*       *        *     *           ***             * *
*       *       *       *         *****           *   *
*       *       *       *           *            *     *
*       *       *       *           *           *       *
*       *       *       *           *            *     *
*       *       *       *           *             *   *
*       *        *     *            *              * *
*********          ***              *                *
```

2.22 Che cosa visualizzerà il codice seguente?

```
printf("*\n**\n***\n****\n*****\n");
```

2.23 Scrivete un programma che legga cinque interi e quindi determini e visualizzi quelli che, all'interno del gruppo, sono il maggiore e il minore. Usate soltanto le tecniche di programmazione che avete appreso in questo capitolo.

2.24 Scrivete un programma che legga un intero e determini e visualizzi se sia pari o dispari. [*Suggerimento*: usate l'operatore resto. Un numero pari è un multiplo di due. Ogni multiplo di due dà un resto uguale a zero, quando è diviso per 2.]

2.25 Visualizzate le vostre iniziali in stampatello in direzione del fondo della pagina. Costruite ogni lettera in stampatello utilizzando lo stesso carattere che essa rappresenta, come riportato nella pagina seguente:

```
PPPPPPPPP
    P   P
    P   P
    P   P
     P P

  JJ
 J
J
 J
  JJJJJJJ

DDDDDDDDD
D        D
D        D
 D      D
  DDDDD
```

2.26 Scrivete un programma che legga due interi e determini e visualizzi se il primo sia un multiplo del secondo. [Suggerimento: usate l'operatore resto.]

2.27 Visualizzate il disegno di una scacchiera utilizzando otto istruzioni `printf` e quindi stampate lo stesso disegno, con il minor numero possibile d'istruzioni `printf`.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 Distinguete tra i termini errore fatale ed errore non fatale. Perché potreste preferire di incorrere in un errore fatale, invece che in uno non fatale?

2.29 Diamo ora un'occhiata in avanti. In questo capitolo avete appreso degli interi e del tipo `int`. Il C può rappresentare anche le lettere maiuscole, quelle minuscole e una considerevole varietà di simboli speciali. Il C usa internamente degli interi di un byte per rappresentare ogni singolo carattere. L'insieme dei caratteri usati da un computer e la corrispondente rappresentazione intera per quei caratteri è l'insieme dei caratteri di quel computer. Potrete visualizzare l'intero equivalente della lettera maiuscola `A`, per esempio, eseguendo l'istruzione

```
printf("%d", 'A');
```

Scrivete un programma che visualizzi gli interi equivalenti ad alcune lettere maiuscole, minuscole, numeri e simboli speciali. Determinate, come minimo, l'intero equivalente di: `A B C a b c 0 1 2 $ * + /` e del carattere spazio.

2.30 Scrivete un programma che prenda in input un numero di cinque cifre, lo spezzetti nelle sue singole cifre e le visualizzi ognuna separata dall'altra da tre spazi. [*Suggerimento*: usate in modo combinato gli operatori di divisione intera e resto.] Per esempio, se l'utente digitasse `42339`, il programma dovrebbe visualizzare

```
4   2   3   3   9
```

2.31 Usando soltanto le tecniche che avete appreso in questo capitolo, scrivete un programma che calcoli i quadrati e i cubi dei numeri da 0 a 10 e utilizzi le tabulazioni per visualizzare la seguente tabella di valori:

```
numero      quadrato    cubo
0           0           0
1           1           1
2           4           8
3           9           27
4           16          64
5           25          125
6           36          216
7           49          343
8           64          512
9           81          729
10          100         1000
```

Capitolo 3

Lo sviluppo di programmi strutturati in C

Obiettivi

- Comprendere le tecniche fondamentali per la risoluzione dei problemi.
- Essere in grado di sviluppare algoritmi, utilizzando il processo top-down per raffinamenti successivi.
- Essere in grado di utilizzare i comandi di selezione `if` e `if/else`, per selezionare le azioni.
- Essere in grado di utilizzare il comando di iterazione `while`, per eseguire ripetutamente delle istruzioni, in un programma.
- Comprendere i cicli controllati da un contatore, o da un valore sentinella.
- Comprendere la programmazione strutturata.
- Essere in grado di utilizzare gli operatori di incremento, di decremento e di assegnamento.

3.1 Introduzione

Prima di scrivere un programma che risolva un particolare problema, è essenziale avere una piena comprensione di quest'ultimo e un approccio pianificato con cura per risolverlo. I prossimi due capitoli tratteranno delle tecniche che facilitano lo sviluppo di programmi strutturati per computer. Nella Sezione 4.12, presenteremo un sommario della programmazione strutturata, che metterà insieme le tecniche sviluppate in questo e nel Capitolo 4.

3.2 Gli algoritmi

La risoluzione di ogni problema di elaborazione comporta l'esecuzione, in un ordine specifico, di una serie di azioni. Una *procedura* che risolva un problema in termini di

1. *azioni* che devono essere eseguite e
2. l'*ordine* in cui tali azioni devono essere eseguite

è detta *algoritmo*. L'esempio seguente dimostra che è importante specificare correttamente l'ordine in cui le azioni devono essere eseguite.

Considerate l'algoritmo "sorgi e splendi" adottato da un giovane dirigente per alzarsi dal letto e andare a lavorare:

Alzarsi dal letto.

Togliersi il pigiama.

Fare una doccia.

Vestirsi.

Fare colazione.

Prendere l'auto per recarsi al lavoro.

Questa procedura porta al lavoro il dirigente, ben preparato per prendere delle decisioni critiche. Supponete, tuttavia, che quegli stessi passi siano eseguiti in un ordine leggermente diverso:

Alzarsi dal letto.

Togliersi il pigiama.

Vestirsi.

Fare una doccia.

Fare colazione.

Prendere l'auto per recarsi al lavoro.

In questo caso, il nostro giovane dirigente giungerebbe al lavoro completamente inzuppato. In un programma per computer, la specificazione dell'ordine in cui le istruzioni devono essere eseguite è chiamata *controllo del programma*. In questo e nel prossimo capitolo, esamineremo con cura le possibilità offerte dal C per il controllo del programma.

3.3 Lo pseudocodice

Uno *pseudocodice* è un linguaggio artificiale e informale, che aiuta i programmatori a sviluppare gli algoritmi. Lo pseudocodice che presentiamo in questo contesto è particolarmente utile per lo sviluppo di algoritmi che saranno convertiti in programmi C strutturati. Uno pseudocodice è un linguaggio simile all'italiano di tutti i giorni; è pratico e maneggevole, sebbene non sia realmente un vero linguaggio di programmazione.

In realtà, i programmi scritti in pseudocodice non possono essere eseguiti sui computer. Essi, piuttosto, aiutano il programmatore a "riflettere" sul programma, prima che provi a scriverlo in un linguaggio di programmazione, come il C. Forniremo molti esempi, in questo capitolo, per mostrare come uno pseudocodice possa essere efficacemente utilizzato nello sviluppo di programmi C strutturati.

Uno pseudocodice è composto semplicemente da caratteri, perciò i programmatori potranno immettere in un computer i programmi in pseudocodice, utilizzando confortevolmente un editor di testo. Il computer potrà visualizzare o stampare, a richiesta, una copia aggiornata di un programma in pseudocodice. Questo, se preparato con cura, potrà facilmente essere convertito in un corrispondente programma C. In molti casi, tutto ciò sarà ottenuto sostituendo semplicemente le istruzioni in pseudocodice con quelle equivalenti nel linguaggio C.

Uno pseudocodice consiste unicamente di istruzioni di azione: ovverosia, quelle che saranno eseguite, quando il programma sarà stato convertito dallo pseudocodice al C e sarà stato eseguito in C. Le dichiarazioni non sono istruzioni eseguibili. Queste sono messaggi per il compilatore. Per esempio, la dichiarazione

```
int i;
```

indica semplicemente il tipo di dato della variabile `i`, al compilatore, e induce quest'ultimo a riservare uno spazio in memoria per la suddetta variabile. Questa dichiarazione, dunque, non provocherà l'esecuzione di nessuna azione (come un input, un output, o un calcolo), quando il programma sarà eseguito. Alcuni programmatori scelgono di elencare ogni variabile e di menzionare brevemente gli scopi di ognuna di queste, all'inizio di un programma in pseudocodice. D'altronde, uno pseudocodice è un aiuto informale allo sviluppo dei programmi.

3.4 Le strutture di controllo

Normalmente, le istruzioni presenti in un programma sono eseguite, una dopo l'altra, nell'ordine in cui sono state scritte. Parleremo perciò di *esecuzione sequenziale.* Varie istruzioni del C, delle quali discuteremo presto, consentono al programmatore di specificare che la successiva istruzione da eseguire possa essere diversa da quella effettivamente susseguente nella sequenza. Questo tipo di indicazione è detta *trasferimento del controllo.*

Durante gli anni '60, divenne evidente che l'utilizzo indiscriminato dei trasferimenti di controllo era la causa prima di una gran quantità di difficoltà sperimentate nei gruppi per lo sviluppo del software. L'indice dell'accusa fu puntato sulla *istruzione* `goto`, che consentiva al programmatore di specificare un trasferimento di controllo verso una di un vastissimo raggio di possibili destinazioni all'interno di un programma. La nozione della cosiddetta *programmazione strutturata* divenne quasi un sinonimo di "*eliminazione dei* `goto`".

La ricerca di Böhm e Jacopini (Böhm, C., e G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM,* Vol. 9, No. 5, May 1966, pp. 336-371) ha dimostrato che i programmi possono essere scritti senza usare nemmeno una istruzione `goto`. La sfida dell'epoca, per i programmatori, divenne quella di cambiare il proprio stile in una "programmazione senza `goto`". Ma fu solo nel pieno degli anni '70 che una gran quantità di programmatori cominciò a prendere seriamente in considerazione la programmazione strutturata. I risultati sono stati davvero impressionanti, dato che i gruppi di sviluppo ottennero: tempi di produzione ridotti, puntualità più frequente nel consegnare i sistemi, e una maggior frequenza di completamenti dei progetti software nei termini previsti dal budget. La chiave di questi successi è semplicemente il fatto che, con le tecniche strutturate, i programmi prodotti sono più chiari, facili da mettere a punto e modificare e, innanzi tutto, con maggiori probabilità di essere esenti da errori.

Il lavoro di Böhm e Jacopini ha dimostrato che tutti i programmi possono essere scritti in termini di tre sole *strutture di controllo*: la *struttura di sequenza*, la *struttura di selezione*, e la *struttura di iterazione.* La struttura di sequenza è implicita in C. Sempre che non gli si ordini diversamente, il computer esegue automaticamente le istruzioni del C, una dopo l'altra, nell'ordine in cui sono state scritte. Il segmento di *diagramma di flusso (flowchart)* nella Figura 3.1 illustra la struttura di sequenza del C.

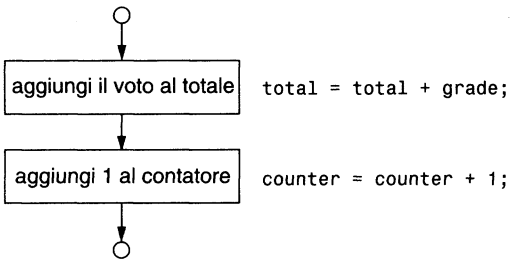


Figura 3.1 Il diagramma di flusso della struttura di sequenza del C

Un diagramma di flusso è una rappresentazione grafica di un algoritmo o di una porzione di questo. I diagrammi di flusso sono disegnati usando certi simboli con significati speciali, come i rettangoli, i rombi, gli ovali e i cerchietti; questi simboli sono connessi tra loro da frecce chiamate *linee di flusso*.

Allo stesso modo dello pseudocodice, sebbene questo sia preferito da molti programmatori, i diagrammi di flusso sono utili per sviluppare e rappresentare gli algoritmi. I diagrammi di flusso mostrano con chiarezza il modo in cui operano le strutture di controllo; in questo testo li useremo solo per tale scopo.

Considerate il segmento di diagramma di flusso per la struttura sequenziale mostrato nella Figura 3.1. Usiamo il *simbolo rettangolo*, chiamato anche *simbolo di azione*, per indicare ogni tipo di attività che includa un calcolo o una operazione di input/output. Le linee di flusso presenti nella figura indicano l'ordine in cui le azioni dovranno essere eseguite: in primo luogo, `grade` dovrà essere aggiunto a `total` e quindi `1` dovrà essere sommato a `counter`. Il C ci consente di avere, in una struttura di sequenza, tante azioni quante ne vogliamo. Come vedremo presto, potranno essere inserite molte istruzioni in sequenza laddove ne possa essere sistemata una.

Nel disegnare un diagramma di flusso che rappresenti un algoritmo *completo*, il primo a essere utilizzato sarà il *simbolo ovale* contenente la parola "Inizio"; mentre un simbolo ovale, contenente la parola "Fine", sarà l'ultimo simbolo utilizzato. Nel disegnare solo una porzione di un algoritmo, come nella Figura 3.1, i simboli ovali saranno omessi, in favore dell'utilizzo di *simboli cerchietto* detti anche *simboli di connessione*.

Il simbolo più importante dei diagrammi di flusso è forse il *simbolo rombo*, detto anche *simbolo di decisione*, indicante che dovrà essere eseguita una scelta. Discuteremo del simbolo rombo nella prossima sezione.

Il C fornisce tre tipi di strutture di selezione sotto forma di comandi. Il comando di selezione `if` (Sezione 3.5), nel caso che una data condizione sia vera, eseguirà (selezionerà) una certa azione, mentre la ignorerà, nel caso che la condizione sia falsa. Il comando di selezione `if...else` (Sezione 3.6) eseguirà una certa azione, nel caso che una data condizione sia vera, mentre ne eseguirà una differente, nel caso che la suddetta condizione sia falsa. Il comando di selezione `switch` (discusso nel Capitolo 4) sceglierà ed eseguirà una tra tante differenti azioni, secondo il valore assunto da una data espressione. Il comando `if` è detto *comando di selezione singola*, perché seleziona o ignora una singola azione. Il comando `if...else` è detto *comando di selezione doppia*, perché seleziona una di due differenti azioni. Il comando `swicth` è detto *comando di selezione multipla*, perché seleziona una tra varie differenti azioni.

Il C fornisce tre tipi di strutture di iterazione sotto forma di comandi: il `while` (Sezione 3.7), il `do...while` e il `for` (entrambi discussi nel Capitolo 4).

Questo è tutto. Il C ha soltanto sette comandi di controllo: la sequenza, tre tipi di selezione e altrettanti di iterazione. Ogni programma C è formato dalla combinazione di ognuno dei suddetti comandi di controllo, utilizzati in quantità appropriate al programma che implementa l'algoritmo. Vedremo che la rappresentazione come diagramma di flusso di ogni comando di controllo, come quello sequenziale della Figura 3.1, ha due simboli cerchietto, uno nel punto di entrata e l'altro in quello di uscita. Questi *comandi di controllo con un ingresso e una uscita singoli* semplificano la costruzione dei programmi. I segmenti di diagrammi di flusso dei comandi di controllo possono essere attaccati l'uno all'altro, collegando il punto di uscita di uno di essi con quello di entrata del successivo. Tutto ciò è simile al modo in cui i bambini innestano i mattoncini delle costruzioni, perciò chiamiamo questo modo di operare *accatastamento dei comandi di controllo*. Impareremo che c'è soltanto un altro modo in cui i comandi di controllo potranno essere innestati: ovverosia, un metodo chiamato *nidificazione dei comandi di controllo*. Di conseguenza, qualsiasi programma C, di cui potremmo aver bisogno, potrà essere costruito utilizzando solamente sette tipi di comandi di controllo, combinati in due soli modi. Questa è l'essenza della semplicità.

3.5 Il comando di selezione if

Le strutture di selezione sono usate per scegliere tra percorsi di azione alternativi. Per esempio, supponete che la votazione per superare un esame sia 60. L'istruzione in pseudocodice

Se il voto dello studente è maggiore o uguale a 60
Visualizza «Promosso»

determina se la condizione "voto dello studente è maggiore o uguale a 60" è vera o falsa. Nel caso in cui la condizione sia vera, sarà visualizzato "Promosso" ed "eseguita" l'istruzione successiva, nell'ordine indicato dallo pseudocodice (ricordate che questo non è un vero linguaggio di programmazione). Nel caso in cui la condizione sia falsa, la visualizzazione sarà ignorata e sarà eseguita l'istruzione successiva, nell'ordine indicato dallo pseudocodice. Osservate che la seconda riga di questa struttura di selezione è rientrata. Tale rientro è opzionale, ma è altamente raccomandato, poiché aiuta a dare risalto alla struttura intrinseca dei programmi. Applicheremo con molta cura le convenzioni dei rientri in tutto il libro. Il compilatore C ignora i *caratteri di spazio bianco*, come gli spazi, le tabulazioni e le newline utilizzate per i rientri e la spaziatura verticale.

Buona abitudine 3.1

Applicare le convenzioni dei rientri, in modo consistente e responsabile, migliora enormemente la leggibilità dei programmi. Per i rientri suggeriamo una tabulazione fissa di 1/4 di pollice (6,35mm) o di tre spazi. In questo libro useremo tabulazioni di tre spazi per ogni rientro.

La precedente istruzione *Se* in pseudocodice può essere scritta in C come

```
if ( grade >= 60 )
   printf( "Promosso\n" );
```

Osservate che il codice C corrisponde strettamente allo pseudocodice. Questa è una delle proprietà dello pseudocodice che lo rende un utile strumento di sviluppo dei programmi.

Buona abitudine 3.2

Lo pseudocodice è spesso utilizzato per «riflettere» su un programma, durante la fase di progettazione dello stesso. Il programma in pseudocodice sarà successivamente convertito in C.

Il diagramma di flusso della Figura 3.2 illustra il comando di selezione singola `if`. Questo contiene quello che è forse il simbolo più importante dei diagrammi di flusso: il *simbolo rombo*, detto anche *simbolo di decisione*, il quale indica che dovrà essere effettuata una scelta. Il simbolo di decisione contiene una espressione, per esempio una condizione, che può essere vera o falsa. Il simbolo di decisione ha due linee di flusso che emergono dallo stesso. Una indica la direzione da seguire quando l'espressione all'interno del simbolo è vera; l'altra indica la direzione da seguire quando l'espressione è falsa. Nel Capitolo 2, abbiamo appreso che le decisioni possono essere prese secondo delle condizioni, che contengono operatori relazionali o di uguaglianza. In realtà, una scelta potrà essere compiuta basandosi su qualsiasi espressione: se essa è valutata zero, sarà trattata come falsa, mentre se il suo valore è diverso da zero, sarà considerata vera.

Osservate che anche il comando `if` è una struttura con un ingresso e una uscita singoli. Apprenderemo presto che anche i diagrammi di flusso per le rimanenti strutture di controllo contengono (a parte i cerchietti e le linee di flusso) soltanto dei simboli rettangolo, per indicare le azioni da eseguire, e dei simboli rombo, per indicare le decisioni da prendere. Questo è il modello di programmazione basato su azioni e decisioni che abbiamo evidenziato.

Possiamo immaginare sette contenitori, ognuno dei quali contiene solo uno tra i sette tipi di diagrammi di flusso dei comandi di controllo. Tali segmenti di diagrammi di flusso sono vuoti: non è scritto niente nei rettangoli e altrettanto nei rombi. Il compito del programmatore è quindi quello di assemblare un programma, con tanti comandi di controllo di ogni tipo quanti ne richiede l'algoritmo, combinandoli in due possibili modi (ovverosia impilandoli o nidificandoli) e completandoli, in un secondo momento, con le azioni e le decisioni appropriate all'algoritmo. Discuteremo in seguito le varietà dei modi in cui possono essere scritte le azioni e le decisioni.

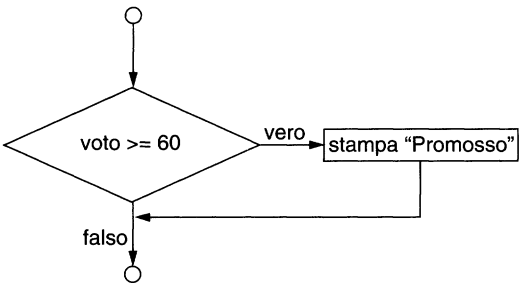


Figura 3.2 Il diagramma di flusso del comando di selezione singola if

3.6 Il comando di selezione if...else

Il comando di selezione `if` esegue l'azione indicata solo quando la condizione è vera; in caso contrario, l'azione è ignorata. Il comando di selezione `if...else` consente al programmatore di specificare che, nel caso in cui la condizione sia vera, dovrà essere eseguita una azione

differente da quella che si dovrà eseguire qualora la condizione sia falsa. Per esempio, l'istruzione in pseudocodice

Se il voto dello studente è maggiore o uguale a 60
    Visualizza «Promosso»
altrimenti
    Visualizza «Bocciato»

visualizza *Promosso* se la votazione dello studente è maggiore o uguale a 60, mentre visualizza *Bocciato* se il voto dello studente è inferiore a 60. In entrambi i casi, dopo la visualizzazione, sarà "eseguita" l'istruzione successiva nella sequenza dello pseudocodice. Osservate che anche il corpo dell'*altrimenti* è rientrato.

Buona abitudine 3.3

Fate rientrare entrambe le istruzioni del corpo di un comando `if...else`.

Qualunque convenzione di rientri abbiate scelto dovrà essere applicata con cura nei vostri programmi. È difficile leggere un programma che non obbedisca in modo uniforme alle convenzioni di spaziatura.

Buona abitudine 3.4

Nel caso dovessero esserci diversi livelli di rientri, per ognuno di essi impiegare lo stesso quantitativo di spazio.

La precedente istruzione *Se/altrimenti* in pseudocodice può essere scritta in C come

```
if ( grade >= 60 )
   printf( "Promosso\n" );
else
   printf( "Bocciato\n" );
```

Il diagramma della Figura 3.3 illustra esattamente il flusso di controllo per il comando `if...else`. Ancora una volta, osservate che, a parte i cerchietti e le frecce, gli unici simboli presenti nel diagramma di flusso sono i rettangoli (per le azioni) e il rombo (per la decisione). Continuiamo a evidenziare il modello di elaborazione basato su azioni e decisioni. Immaginate nuovamente un contenitore capiente, riempito con tanti comandi vuoti di selezione doppia (rappresentati da segmenti di diagrammi di flusso) quanti ne possono servire per costruire qualsiasi programma C. Il lavoro del programmatore, ripetiamo, è quello di assemblare i suddetti comandi di selezione, accatastandoli e nidificandoli, con qualsiasi altro comando di controllo possa essere richiesto dall'algoritmo, e completando i rettangoli e i rombi vuoti, con le azioni e le decisioni appropriate per l'algoritmo che si sta implementando.

Il C fornisce l'*operatore condizionale* (`?:`) che è strettamente correlato con il comando `if...else`. Quello condizionale è l'unico *operatore ternario* del C: in altre parole, accetta tre operandi. Gli operandi, insieme all'operatore condizionale, formano una *espressione condizionale*. Il primo operando è una condizione, il secondo operando è il valore che assumerà l'intera espressione condizionale, qualora la condizione sia vera, mentre il terzo operando è il valore che assumerà l'intera espressione condizionale, qualora la condizione sia falsa. Per esempio, l'istruzione `printf`

```
printf( "%s\n", grade >= 60 ? "Promosso" : "Bocciato" );
```

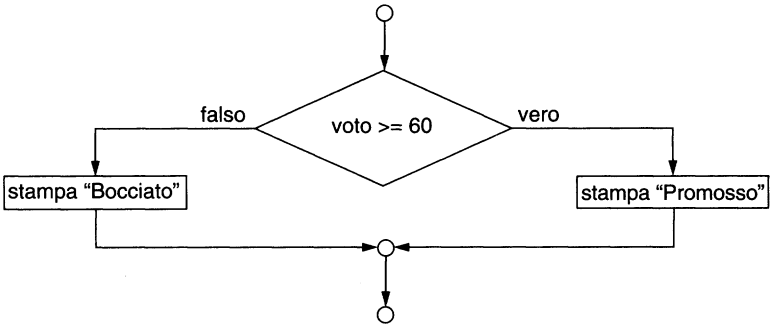


Figura 3.3 Il diagramma di flusso del comando di selezione doppia if...else

contiene un'espressione condizionale che restituirà la stringa letterale `"Promosso"`, qualora la condizione `grade >= 60` risulti vera, mentre restituirà la stringa letterale `"Bocciato"`, qualora la condizione risulti falsa. La stringa di controllo del formato per la `printf` contiene la specifica di conversione `%s` per visualizzare una stringa di caratteri. Di conseguenza, l'istruzione `printf` precedente eseguirà essenzialmente la stessa operazione dell'istruzione `if...else` precedente.

I valori in una espressione condizionale possono anche essere delle azioni da eseguire. Per esempio, l'espressione condizionale

```
grade >= 60 ? printf( "Promosso\n" ) : printf( "Bocciato\n" );
```

si leggerà, "Se `grade` è maggiore o uguale a `60` allora `printf( "Promosso\n" )`, altrimenti `printf( "Bocciato\n" )`". Anche questa espressione condizionale è comparabile con il comando `if...else` precedente. Vedremo che gli operatori condizionali potranno essere utilizzati in alcune situazioni dove non potranno essere utilizzate le istruzioni `if...else`.

I comandi *if...else nidificati* sono in grado di verificare diversi casi, se si sistemano degli `if...else` all'interno di altri comandi dello stesso tipo. Per esempio, la seguente istruzione in pseudocodice, visualizzerà `A` per votazioni di esame maggiori o uguali a `90`, `B` per voti maggiori o uguali a `80`, `C` per voti maggiori o uguali a `70`, `D` per votazioni maggiori o uguali a `60`, `F` per tutte le altre votazioni.

```
Se il voto dello studente è maggiore o uguale a 90
   Visualizza «A»
altrimenti
   Se il voto dello studente è maggiore o uguale a 80
      Visualizza «B»
   altrimenti
      Se il voto dello studente è maggiore o uguale a 70
         Visualizza «C»
      altrimenti
         Se il voto dello studente è maggiore o uguale a 60
            Visualizza «D»
```

altrimenti

Visualizza «F»

Questo pseudocodice può essere scritto in C come

```
if ( grade >= 90 )
   printf( "A\n" );
else
   if ( grade >= 80 )
      printf( "B\n" );
   else
      if ( grade >= 70 )
         printf( "C\n" );
      else
         if ( grade >= 60 )
            printf( "D\n" );
         else
            printf( "F\n" );
```

Qualora la variabile `grade` sia maggiore o uguale a `90`, saranno vere le prime quattro condizioni, ma sarà eseguita soltanto l'istruzione `printf` successiva al primo controllo. Dopo che quella `printf` sarà stata eseguita, la parte `else` dell'istruzione `if...else` "più esterna" sarà saltata. Molti programmatori C preferiscono scrivere il precedente comando `if` come

```
if ( grade >= 90 )
   printf( "A\n" );
else if ( grade >= 80 )
   printf( "B\n" );
else if ( grade >= 70 )
   printf( "C\n" );
else if ( grade >= 60 )
   printf( "D\n" );
else
   printf( "F\n" );
```

Per come è concepito il compilatore C, entrambe le forme sono equivalenti. L'ultima forma è più diffusa perché evita un rientro del codice troppo profondo verso il margine destro. Un tale rientro lascia spesso poco spazio sulle righe, costringendo alla frammentazione delle stesse e facendo diminuire la leggibilità del programma.

Il comando di selezione `if` si aspetta una sola istruzione nel proprio corpo. Per includere più istruzioni nel corpo di un `if`, racchiudete tra parentesi graffe ({ e }) il gruppo di istruzioni. Un gruppo di istruzioni contenuto all'interno di una coppia di parentesi graffe è una *istruzione composta* o un *blocco*.

Ingegneria del software 3.1

Una istruzione composta può essere inserita in qualsiasi parte del programma in cui possa essere sistemata una istruzione singola.

L'esempio successivo include una istruzione composta nella parte `else` di un comando `if...else`.

```
if ( grade >= 60 )
   printf( "Promosso.\n" );
else {
   printf( "Bocciato.\n" );
   printf( "Devi ripetere questo corso.\n" );
}
```

In questo caso, qualora la votazione fosse inferiore a `60`, il programma eseguirebbe entrambe le istruzioni `printf` presenti nel corpo di **`else`** e visualizzerebbe

```
Bocciato.
Devi ripetere questo corso.
```

Osservate le parentesi graffe che circondano le due istruzioni della clausola `else`. Queste parentesi graffe sono importanti. Senza di esse, l'istruzione

```
printf("Devi ripetere questo corso.\n");
```

sarebbe esterna al ramo `else` dell'istruzione `if` e verrebbe eseguita, senza considerare se la votazione sia o no inferiore a `60`.

Errore tipico 3.1

Dimenticare una o entrambe le parentesi graffe che delimitano una istruzione composta.

Un errore di sintassi è intercettato dal compilatore. Un errore logico ha il suo effetto durante l'esecuzione. Un errore logico fatale conduce al fallimento e alla terminazione prematura di un programma. Un errore logico non fatale consente al programma di continuare l'esecuzione ma anche di produrre risultati errati.

Errore tipico 3.2

Inserire un punto e virgola, dopo la condizione di un comando `if`*, condurrebbe a un errore logico nei comandi di selezione singola* `if`*, mentre provocherebbe un errore di sintassi nei comandi di selezione doppia* `if`...`else`*.*

Collaudo e messa a punto 3.1

Digitare la parentesi graffa iniziale e quella finale di una istruzione composta, prima di digitare le singole istruzioni all'interno delle parentesi graffe, aiuta a evitare di omettere una o entrambe le parentesi graffe, evitando errori di sintassi ed errori logici (laddove entrambe le parentesi siano di fatto necessarie).

Ingegneria del software 3.2

Proprio come una istruzione composta potrà essere inserita laddove potrebbe essere sistemata una istruzione singola, sarà anche possibile non averne per nulla, in altre parole, avere l'istruzione vuota. L'istruzione vuota è rappresentata dall'inserimento di un punto e virgola (;) laddove normalmente ci sarebbe stata una istruzione.

3.7 Il comando di iterazione while

Un *comando di iterazione* consente al programmatore di specificare che una azione dovrà essere ripetuta finché alcune condizioni rimarranno vere. L'istruzione in pseudocodice

Finché ci sono ancora articoli nella mia lista della spesa
Compra il prossimo articolo e cancellalo dalla mia lista

descrive l'iterazione che si verifica durante una passeggiata per gli acquisti. La condizione "ci sono ancora articoli nella mia lista della spesa" potrà essere vera o falsa. Se sarà vera, allora sarà eseguita l'azione "Compra il prossimo articolo e cancellalo dalla mia lista". Questa azione sarà eseguita ripetutamente fintanto che la condizione rimarrà vera. Le istruzioni contenute nel comando di iterazione *finché* costituiscono il corpo dello stesso. Questo potrà essere formato da una istruzione singola oppure da una composta.

Infine, la condizione diventerà falsa (quando l'ultimo articolo della lista sarà stato acquistato e cancellato dalla stessa). A questo punto, il ciclo potrà terminare e sarà eseguita la prima istruzione dello pseudocodice successiva alla struttura di iterazione.

Errore tipico 3.3

Non fornire, nel corpo di un comando `while`*, una azione che faccia eventualmente diventare falsa la condizione del* `while`*. Normalmente, una tale struttura di iterazione non avrà mai fine: un errore chiamato «ciclo infinito».*

Errore tipico 3.4

Scrivere la parola chiave `while` *con una* `W` *maiuscola come in* `While` *(ricordate che il C è un linguaggio che distingue le maiuscole dalle minuscole). Tutte le parole chiave riservate del C come* `while`*,* `if` *ed* `else` *contengono soltanto lettere minuscole.*

Consideriamo ora, come esempio di un `while` reale, un segmento di programma progettato per trovare la prima potenza di due maggiore di 1000. Supponete che la variabile intera `product` sia stata inizializzata con 2. Nel momento in cui il seguente comando di iterazione `while` avrà terminato la propria esecuzione, `product` conterrà la risposta desiderata:

```
product = 2;

while ( product <= 1000 )
   product = 2 * product;
```

Il diagramma della Figura 3.4 illustra precisamente il flusso di controllo per il comando di iterazione `while`. Ancora una volta, osservate che, a parte i cerchietti e le frecce, il diagramma di flusso contiene soltanto un rettangolo e un rombo. Il diagramma di flusso mostra chiaramente l'iterazione. La linea di flusso emergente dal rettangolo torna indietro alla decisione, che sarà verificata ogni volta all'interno del ciclo, finché la stessa non diventerà eventualmente falsa. A questo punto, il comando `while` avrà completato il suo compito e il controllo passerà all'istruzione successiva del programma.

Al momento dell'ingresso nel comando `while`, il valore di `product` era 2. La variabile `product` sarà moltiplicata ripetutamente per 2, assumendo nel tempo i valori 4, 8, 16, 32, 64, 128, 256, 512 e 1024. Nel momento in cui `product` avrà assunto il valore 1024, la condizione nel comando `while`, `product <= 1000`, sarà diventata falsa. Ciò provocherà il termine del ciclo e il valore finale di `product` sarà 1024. L'esecuzione del programma continuerà con la prossima istruzione dopo il `while`.

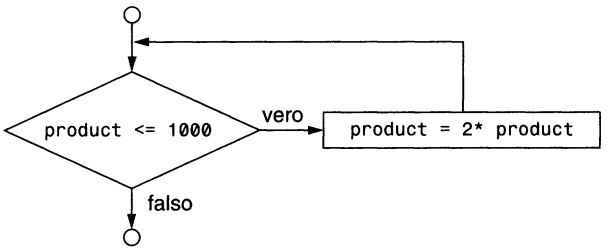


Figura 3.4 Il diagramma di flusso del comando di iterazione while

3.8 Formulazione degli algoritmi: studio di un caso 1 (iterazione controllata da un contatore)

Per mostrare come sono sviluppati gli algoritmi, risolveremo diverse varianti di un problema per il calcolo della media di una classe. Considerate la seguente enunciazione del problema:

Una classe di dieci studenti sostiene un esame. Avete a disposizione le votazioni (degli interi nell'intervallo da 0 a 100) per questo esame. Determinate la media della classe in questo esame.

La media della classe sarà uguale alla somma delle votazioni divisa per il numero degli studenti. L'algoritmo per risolvere questo problema su un computer dovrà prendere in input ognuna delle votazioni, eseguire il calcolo della media e visualizzare il risultato.

Usiamo lo pseudocodice, elenchiamo le azioni che dovranno essere eseguite e specifichiamo l'ordine in cui queste azioni dovranno essere eseguite. Useremo una *iterazione controllata da un contatore*, per prendere in input una per volta le votazioni. Questa tecnica utilizza una variabile detta *contatore*, per specificare il numero di volte che un insieme di istruzioni dovrà essere eseguito. In questo esempio, l'iterazione terminerà quando il contatore avrà superato il valore 10. In questa sezione, presenteremo solamente lo pseudocodice per l'algoritmo (Figura 3.5) e il corrispondente programma C (Figura 3.6). Nella prossima sezione, mostreremo come vengono sviluppati gli algoritmi in pseudocodice. Una iterazione controllata da un contatore è spesso chiamata *iterazione definita*, poiché il numero delle iterazioni è noto prima che il ciclo incominci la propria esecuzione.

Osservate nell'algoritmo i riferimenti a un totale e a un contatore. Un *totale* è una variabile utilizzata per accumulare la somma di una serie di valori. Un contatore è una variabile utilizzata per contare; in questo caso, per contare il numero dei voti immesso. Le variabili usate per immagazzinare i totali normalmente devono essere azzerate, prima di poter essere utilizzate in un programma; altrimenti la somma includerebbe il valore immagazzinato in precedenza nella locazione di memoria del totale. Secondo il loro utilizzo, le variabili contatore sono normalmente inizializzate con zero o uno (presenteremo degli esempi che mostrano ognuno di questi utilizzi). Una variabile non inizializzata contiene un *valore "immondizia"*: l'ultimo dato immagazzinato nella locazione di memoria riservata per quella variabile.

Errore tipico 3.5

Nel caso in cui un contatore o un totale non sia stato inizializzato, probabilmente i risultati del vostro programma non saranno corretti. Questo è un esempio di errore logico.

Inizializzare il totale a zero
Inizializzare il contatore dei voti a uno

Finché il contatore resta minore o uguale a dieci
    Prendere dall'input il prossimo voto
    Aggiungere il voto al totale
    Aggiungere uno al contatore

Impostare il valore della media al totale diviso dieci
Visualizzare la media

Figura 3.5 L'algoritmo in pseudocodice che utilizza una iterazione controllata da un contatore, per risolvere il problema del calcolo della media di una classe

```
1  /* Fig. 3.6: fig03_06.c
2     Programma per il calcolo della media di una classe con una iterazione
      controllata da un contatore */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int counter; /* numero di voti da inserire in seguito */
9     int grade; /* valore del voto */
10    int total; /* somma dei voti inseriti dall'utente */
11    int average; /* media dei voti */
12
13    /* fase di inizializzazione */
14    total = 0; /* inizializza il totale */
15    counter = 1; /* inizializza il contatore del ciclo */
16
17    /* fase di elaborazione */
18    while ( counter <= 10 ) { /* itera 10 volte */
19       printf( "Enter grade: " ); /* sollecita l'inserimento */
20       scanf( "%d", &grade ); /* legge il voto inserito dall'utente */
21       total = total + grade; /* aggiunge il voto al totale */
22       counter = counter + 1; /* incrementa il contatore */
23    } /* fine del comando while */
24
25    /* fase di chiusura */
26    average = total / 10; /* divisione fra interi */
27
28    printf( "Class average is %d\n", average ); /* visualizza
                                                     il risultato */
29
```

Figura 3.6 Programma C ed esecuzione di esempio per il problema del calcolo della media di una classe, con una iterazione controllata da un contatore (continua)

```
30     return 0;   /* indica che il programma è terminato con successo */
31
32  } /* fine della funzione main */
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Figura 3.6 Programma C ed esecuzione di esempio per il problema del calcolo della media di una classe, con una iterazione controllata da un contatore

Collaudo e messa a punto 3.2

Inizializzate i contatori e i totali.

Osservate che il calcolo della media nel programma ha prodotto il risultato intero 81. In realtà, la somma delle votazioni in questo esempio è 817 che diviso per dieci dovrebbe produrre 81,7, in altre parole, un numero con una virgola decimale. Vedremo come trattare questi numeri (detti in virgola mobile) nella prossima sezione.

3.9 Formulazione degli algoritmi con processo top-down per raffinamenti successivi: studio di un caso 2 (iterazione controllata da un valore sentinella)

Generalizziamo il problema del calcolo della media di una classe. Considerate il seguente problema:

Sviluppate un programma per il calcolo della media di una classe, che elaborerà un numero arbitrario di votazioni ogni volta che il programma sarà eseguito.

Nel primo esempio di calcolo della media di una classe, il numero delle votazioni (10) era noto in anticipo. In questo esempio, non è stata data alcuna indicazione su quante votazioni saranno immesse. Il programma dovrà elaborare un numero arbitrario di votazioni. In quale modo il programma potrà determinare quando terminare l'immissione delle votazioni? In quale modo potrà sapere quando calcolare e visualizzare la media della classe?

Una maniera per risolvere questo problema è di usare un valore speciale, detto *valore sentinella* (o anche *valore di segnalazione, valore dummy (fittizio)* o *valore flag (bandiera)*), per indicare la "fine della immissione dei dati". L'utente immetterà le votazioni finché tutte quelle legittime saranno state immesse. L'utente immetterà quindi il valore sentinella, per indicare che

l'ultima valutazione sarà stata immessa. Le iterazioni controllate da un valore sentinella sono spesso dette *iterazioni indefinite* perché, prima che inizi l'esecuzione del ciclo, il numero delle iterazioni non è noto.

Chiaramente, il valore sentinella dovrà essere scelto in modo che non possa essere confuso con un valore di input accettabile. Dato che le valutazioni di un esame sono normalmente degli interi non negativi, –1 sarà un valore sentinella accettabile per questo problema. Di conseguenza, una esecuzione del programma per il calcolo della media di una classe potrà elaborare un flusso di input come 95, 96, 75, 74, 89 e –1. Il programma quindi calcolerà e visualizzerà la media della classe per le votazioni 95, 96, 75, 74 e 89 (–1 è il valore sentinella, perciò non dovrà essere inserito nel calcolo della media).

Errore tipico 3.6

Scegliere un valore sentinella che sia anche un valore legittimo per i dati.

Affronteremo ora il problema del calcolo della media di una classe, con una tecnica chiamata *top down (dall'alto in basso) per raffinamenti successivi*: una tecnica essenziale per lo sviluppo di programmi ben strutturati. Cominciamo con una rappresentazione in pseudocodice del *top*:

Determinare la media della classe per l'esame

Il top è una singola istruzione che specifica tutte le funzioni del programma. Per come è definito, il top è in effetti una rappresentazione completa di un programma. Sfortunatamente, il top specifica raramente una quantità sufficiente di dettagli da cui ricavare un programma C.

Per questo motivo ora diamo il via al processo di raffinamento. Dividiamo il top in una serie di piccole attività e le elenchiamo nell'ordine in cui dovranno essere eseguite. Ciò produrrà il seguente *primo raffinamento.*

Inizializzare le variabili

Prendere in input, sommare e contare le votazioni dell'esame

Calcolare e visualizzare la media della classe

In questo caso è stata usata soltanto la struttura di sequenza: infatti, i passi elencati dovranno essere eseguiti in ordine, uno dopo l'altro.

Ingegneria del software 3.3

Ogni raffinamento, così come lo stesso top, è una enunciazione completa dell'algoritmo; varia solo il livello di dettaglio.

Per procedere verso il livello successivo di raffinamento, ovverosia il *secondo*, ci impegneremo nella specifica delle variabili. Abbiamo bisogno di un totale progressivo dei numeri, un contatore della quantità di quelli che sono stati elaborati, una variabile per ricevere il valore di ogni valutazione, quando sarà stato immesso nell'input, e una variabile per conservare la media calcolata. L'istruzione in pseudocodice

Inizializzare le variabili

potrà essere raffinata come segue:

Inizializzare il totale a zero

Inizializzare il contatore a zero

Osservate che occorrerà inizializzare solo il totale e il contatore. Le variabili per la media e le valutazioni (rispettivamente, per la media calcolata e l'input dell'utente) non dovranno essere necessariamente inizializzate, perché i loro valori saranno sostituiti dal processo di scrittura distruttiva, di cui abbiamo discusso nel Capitolo 2. L'istruzione in pseudocodice

Prendere in input, sommare e contare le votazioni dell'esame

richiede una struttura di iterazione (ovverosia un ciclo) che prenda in input consecutivamente ogni valutazione. Dato che non sappiamo in anticipo quante valutazioni dovranno essere elaborate, utilizzeremo una iterazione controllata da un valore sentinella. L'utente digiterà una per volta le valutazioni legittime. Una volta che l'ultima valutazione legittima sarà stata digitata, l'utente immetterà il valore sentinella. Per ognuna delle valutazioni immesse, il programma controllerà l'eventuale immissione del valore sentinella e terminerà il ciclo quando l'avrà incontrato. Il raffinamento della precedente istruzione in pseudocodice sarà quindi

Prendere in input la prima valutazione
Finché l'utente non ha ancora immesso il valore sentinella
    Aggiungere questa valutazione al totale corrente
    Aggiungere uno al contatore di valutazioni
    Prendere in input la prossima valutazione (o forse il valore sentinella)

Osservate che, nello pseudocodice, non utilizziamo parentesi graffe intorno all'insieme di istruzioni che formano il corpo del comando *finché*. Facciamo rientrare semplicemente queste istruzioni sotto il *finché*, per mostrare che appartengono tutte a *finché*. Di nuovo, lo pseudocodice è soltanto un aiuto informale allo sviluppo del programma.

L'istruzione in pseudocodice

Calcolare e visualizzare la media della classe

potrà essere raffinata come segue:

Se il contatore non è uguale a zero
    Impostare la media con il totale diviso per il contatore
    Visualizzare la media
altrimenti
    Visualizzare "Non sono state immesse valutazioni"

Osservate che in questo caso siamo stati attenti a verificare la possibilità di una divisione per zero: un *errore fatale* che, se non fosse stato individuato, avrebbe causato il fallimento del programma (detto spesso *bombing* o *crashing*). Il secondo raffinamento completo è mostrato nella Figura 3.7.

Errore tipico 3.7

Un tentativo di divisione per zero genererà un errore fatale.

Buona abitudine 3.5

Quando eseguite una divisione per una espressione che potrebbe assumere un valore uguale a zero, controllate esplicitamente quella evenienza e gestitela in modo appropriato nel vostro programma, per esempio, stampando un messaggio, invece di lasciare che un errore fatale ne interrompa l'esecuzione.

Inizializzare il totale a zero
Inizializzare il contatore a zero

Prendere in input la prima valutazione
Finché l'utente non ha ancora immesso il valore sentinella
    Aggiungere questa valutazione al totale corrente
    Aggiungere uno al contatore di valutazioni
    Prendere in input la prossima valutazione (o forse il valore sentinella)

Se il contatore non è uguale a zero
    Impostare la media con il totale diviso per il contatore
    Visualizzare la media
altrimenti
    Visualizzare "Non sono state immesse valutazioni"

Figura 3.7 L'algoritmo in pseudocodice che utilizza una iterazione controllata da un valore sentinella, per risolvere il problema del calcolo della media di una classe

Nella Figura 3.5 e nella Figura 3.7, per una maggiore leggibilità, abbiamo incluso all'interno dello pseudocodice alcune righe completamente vuote. In realtà, le righe vuote separano questi programmi nelle loro varie fasi.

Ingegneria del software 3.4

Molti programmi possono essere suddivisi logicamente in tre fasi: una fase di inizializzazione, che inizializza le variabili del programma; una fase di elaborazione, che prende in input i valori dei dati e imposta conseguentemente le variabili del programma; una fase di chiusura, che calcola e visualizza i risultati finali.

L'algoritmo in pseudocodice della Figura 3.7 risolve il problema più generale del calcolo della media di una classe. Questo algoritmo è stato sviluppato dopo due soli livelli di raffinamento. A volte saranno necessari più livelli di raffinamento.

Ingegneria del software 3.5

Il programmatore termina il processo top down per raffinamenti successivi, quando l'algoritmo in pseudocodice specificato è sufficientemente dettagliato perché il programmatore possa essere in grado di convertirlo in C. Normalmente, a quel punto, l'implementazione del programma C sarà lineare.

Nella Figura 3.8 sono mostrati il programma C e l'esecuzione di esempio. Nonostante vengano immesse soltanto delle valutazioni intere, il calcolo della media produrrà probabilmente un numero decimale con una virgola decimale. Il tipo di dato `int` non può rappresentare un tale numero. Il programma introduce perciò il tipo di dato `float` per gestire i numeri con una virgola decimale (detti *numeri in virgola mobile*) e un operatore speciale, detto *operatore cast (operatore di conversione)*, per gestire il calcolo della media. Queste caratteristiche saranno spiegate in dettaglio dopo la presentazione del programma.

```
1  /* Fig. 3.8: fig03_08.c
2     Programma per il calcolo della media di una classe con una
      iterazione controllata da un valore sentinella */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int counter; /* numero dei voti inseriti */
9     int grade; /* valore del voto */
10    int total; /* somma dei voti */
11
12    float average; /* numero con la virgola per la media */
13
14    /* fase di inizializzazione */
15    total = 0; /* inizializza il totale */
16    counter = 0; /* inizializza il contatore del ciclo */
17
18    /* fase di elaborazione */
19    /* acquisisce il primo voto dall'utente */
20    printf( "Enter grade, -1 to end: " ); /* sollecita l'inserimento */
21    scanf( "%d", &grade ); /* legge il voto inserito dall'utente */
22
23    /* cicla finché il valore della sentinella non viene inserito
         dall'utente */
24    while ( grade != -1 ) {
25       total = total + grade; /* aggiunge il voto al totale */
26       counter = counter + 1; /* incrementa il contatore */
27
28       /* acquisisce il prossimo voto dall'utente */
29       printf( "Enter grade, -1 to end: " ); /* sollecita l'inserimento */
30       scanf( "%d", &grade ); /* legge il voto inserito dall'utente */
31    } /* fine del ciclo while */
32
33    /* fase di chiusura */
34    /* se l'utente ha inserito almeno un voto */
35    if ( counter != 0 ) {
36
37       /* calcola la media di tutti i voti inseriti */
38       average = ( float ) total / counter; /* evita il troncamento */
39
40       /* visualizza la media con due cifre di precisione */
41       printf( "Class average is %.2f", average );
42    } /* fine del ramo if */
43    else { /* se non è stato inserito alcun voto, visualizza
                un messaggio */
44       printf( "No grades were entered\n" );
```

Figura 3.8 Il programma C e l'esecuzione di esempio per il problema del calcolo della media di una classe, con una iterazione controllata da un valore sentinella (continua)

```
45      } /* fine del ramo else */
46
47      return 0;   /* indica che il programma è terminato con successo */
48
49   } /* fine della funzione main */
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Figura 3.8 Il programma C e l'esecuzione di esempio per il problema del calcolo della media di una classe, con una iterazione controllata da un valore sentinella

Osservate l'istruzione composta nel ciclo `while` (riga 24) della Figura 3.8. Ancora una volta, le parentesi graffe sono necessarie perché tutte e quattro le istruzioni siano eseguite all'interno del ciclo. Senza le parentesi graffe, le tre istruzioni nel corpo del ciclo ricadrebbero all'esterno della iterazione, inducendo il computer a interpretare erroneamente il codice nel modo seguente:

```
while( grade != -1 )
   total = total + grade; /* aggiunge il voto al totale */
counter = counter + 1; /* incrementa il contatore */
printf( "Enter grade, -1 to end: " ); /* sollecita l'inserimento */
scanf( "%d", &grade ); /* legge il voto inserito dall'utente */
```

Ciò provocherebbe un ciclo infinito, qualora l'utente non immettesse –1 come prima valutazione.

Buona abitudine 3.6

In un ciclo controllato da un valore sentinella, il prompt per la richiesta di immissione dei dati dovrà ricordare esplicitamente all'utente qual è il valore sentinella.

Le medie non saranno sempre dei valori interi. Spesso, una media sarà un valore come 7,2 o –93,5 che contengono una parte frazionaria. Questi valori sono indicati come numeri in virgola mobile e sono rappresentati dal tipo di dato `float`. La variabile `average` è dichiarata con il tipo `float` (riga 12) così che possa catturare il risultato frazionario del nostro calcolo. Il risultato del calcolo `total / counter` sarà tuttavia un intero, poiché `total` e `counter` sono entrambe delle variabili intere. Dividere due interi produce una *divisione tra interi* in cui ogni parte frazionaria del calcolo è persa (ovverosia *troncata*). Dato che il calcolo sarà eseguito per primo, la parte frazionaria sarà persa prima che il risultato possa essere assegnato

ad `average`. Per produrre un calcolo in virgola mobile con dei valori interi, dovremo creare dei dati temporanei che siano dei numeri in virgola mobile. Per eseguire questo compito, il C fornisce l'*operatore unario cast (od operatore di conversione)*. La riga 38

```
average = ( float ) total / counter;
```

include l'operatore di conversione `(float)` che creerà una copia temporanea in virgola mobile del suo operando `total`. Il valore immagazzinato in `total` è ancora un intero. L'utilizzo di un operatore di conversione in questo modo è detto *conversione esplicita*. Il calcolo, a questo punto, consisterà di un valore in virgola mobile (la versione `float` temporanea di `total`) diviso per un valore intero immagazzinato in `counter`. Il compilatore C però sa solo valutare espressioni in cui i tipi di dato degli argomenti sono identici. Per assicurarsi che gli operandi siano dello stesso tipo, il compilatore eseguirà su quelli selezionati una operazione detta *promozione* (o anche *conversione implicita*). Per esempio, in una espressione contenente i tipi di dato `int` e `float`, verranno eseguite delle copie degli operandi `int` e queste dovranno essere *promosse* al tipo `float`. Nel nostro esempio, una volta che sarà stata eseguita la copia di `counter` e che questa sarà stata promossa al tipo `float`, sarà eseguito il calcolo e il risultato della divisione in virgola mobile sarà assegnato ad `average`. Il C fornisce un insieme di regole per la promozione dei differenti tipi di operandi. Il Capitolo 5 presenterà una discussione di tutti i tipi di dato standard e del relativo ordine di promozione.

Sono disponibili operatori di conversione per ogni tipo di dato. L'operatore di conversione è composto inserendo delle parentesi tonde intorno al nome del tipo di dato. L'operatore di conversione è un *operatore unario*, ovverosia un operatore che accetta solo un operando. Nel Capitolo 2 abbiamo studiato gli operatori aritmetici binari. Il C supporta anche delle versioni unarie degli operatori più (+) e meno (-), perciò il programmatore potrà scrivere espressioni come `-7` o `+5`. Gli operatori di conversione associano da destra a sinistra e hanno la stessa priorità degli altri operatori unari, come il + e il - unari. Tale priorità è a un livello più alto di quella degli *operatori moltiplicativi* `*`, `/` e `%`.

Il programma nella Figura 3.8 utilizza per la `printf` la specifica di conversione `%.2f` (riga 41) per visualizzare il valore di `average`. La `f` specifica che sarà visualizzato un valore in virgola mobile. Il `.2` è la *precisione* con cui il valore sarà visualizzato. Questa stabilisce che il valore sarà visualizzato con due cifre a destra della virgola dei decimali. Nel caso fosse stata utilizzata la specifica di conversione `%f`, ovverosia senza indicare la precisione, sarebbe stata utilizzata quella *di default* di 6 cifre, proprio come se fosse stata utilizzata la specifica di conversione `%.6f`. Nel momento in cui saranno visualizzati dei valori in virgola mobile con una data precisione, il valore visualizzato sarà *arrotondato* al numero indicato di posizioni decimali. Il valore nella memoria resterà inalterato. Una volta che le seguenti istruzioni saranno state eseguite, saranno visualizzati i valori 3,45 e 3,4.

```
printf( "%.2f\n", 3.446 );    /* visualizza 3,45 */
printf( "%.1f\n", 3.446 );    /* visualizza 3,4 */
```

Errore tipico 3.8

È errato usare la precisione in una specifica di conversione nella stringa per il controllo del formato di una istruzione `scanf`. *Le precisioni possono essere utilizzate soltanto nelle specifiche di conversione di* `printf`.

Errore tipico 3.9

Utilizzare i numeri in virgola mobile, presumendo che siano rappresentati in un modo preciso, potrà condurre a risultati incorretti. I numeri in virgola mobile sono rappresentati dalla maggior parte dei computer solo in modo approssimato.

Collaudo e messa a punto 3.3

Non eseguite un confronto di uguaglianza di valori in virgola mobile.

I numeri in virgola mobile hanno numerose applicazioni, nonostante non siano sempre "precisi al 100%". Per esempio, quando parliamo di una "normale" temperatura corporea di 98,6 gradi Fahrenheit (37 gradi Celsius), non abbiamo bisogno di essere precisi fino a un gran numero di cifre. Nel momento in cui vediamo la temperatura di un termometro e leggiamo 98,6, questa potrebbe essere in realtà 98,5999473210643. La questione è che leggere semplicemente quel numero come 98,6 andrà bene per la maggior parte delle applicazioni. In seguito diremo qualcosa in più a proposito di questo problema.

Un altro modo per mezzo del quale potrebbero essere prodotti dei numeri in virgola mobile è attraverso le divisioni. Quando dividiamo 10 per 3, il risultato è 3,3333333... con la sequenza dei 3 che si ripete all'infinito. Per contenere un tale valore, il computer alloca solamente una quantità prefissata di spazio; è quindi evidente che il valore in virgola mobile immagazzinato potrà essere soltanto una approssimazione.

3.10 Formulazione degli algoritmi con processo top-down per raffinamenti successivi: studio di un caso 3 (strutture di controllo nidificate)

Lavoriamo ora su un altro problema completo. Utilizzando ancora una volta lo pseudocodice e il processo top down per raffinamenti successivi, formuleremo un algoritmo e scriveremo un corrispondente programma C. Abbiamo visto che i comandi di controllo possono essere accatastati l'uno sull'altro (in sequenza), proprio come i bambini accatastano i mattoncini delle costruzioni. In questo studio di un caso, vedremo l'unico altro modo strutturato attraverso il quale i comandi di controllo potranno essere connessi in C: ovverosia attraverso la *nidificazione* di un comando di controllo in un'altro.

Considerate la seguente enunciazione del problema:

Una università offre un corso che prepara gli studenti per l'esame della licenza di stato per agenti immobiliari. Lo scorso anno, molti degli studenti che hanno completato questo corso hanno superato l'esame per ottenere la licenza. Naturalmente, l'università vorrebbe sapere quale sia stato l'esito dei propri studenti al termine dell'esame. Vi è stato chiesto di scrivere un programma che sommi i risultati. Vi è stata anche consegnata una lista di 10 studenti. A fianco a ogni nome è stato scritto un 1, se lo studente ha superato l'esame, e un 2, se lo studente è stato respinto.

Il vostro programma dovrà analizzare i risultati dell'esame come segue:

1. *Prendete in input ogni risultato della prova (ovverosia un 1 o un 2). Visualizzate il messaggio "Enter result" (immettere il risultato) sullo schermo ogni volta che il programma richiede un altro risultato della prova.*

2. *Contate le occorrenze per ogni tipo di risultato della prova.*
3. *Visualizzate un riassunto dei risultati, che indichi il numero degli studenti che hanno superato la prova e quello degli studenti che sono stati respinti.*
4. *Nel caso più di 8 studenti abbiano superato l'esame, visualizzate il messaggio «Raise tuition» (aumentare le tasse).*

Dopo aver letto attentamente l'enunciazione del problema, sviluppiamo le seguenti osservazioni:

1. Il programma dovrà elaborare 10 risultati per la prova. Sarà quindi utilizzato un ciclo controllato da un contatore.
2. Ogni risultato della prova sarà un numero: un 1 o un 2. Ogni volta che leggerà un risultato della prova, il programma dovrà determinare se quel numero sarà stato un 1 o un 2. Nel nostro algoritmo ci limiteremo a controllare il numero 1. Qualora il numero non sia un 1, supporremo che si tratti di un 2. (Un esercizio alla fine del capitolo considererà le conseguenze di questa supposizione).
3. Saranno utilizzati due contatori: uno per contare il numero degli studenti che hanno superato l'esame e l'altro per contare il numero di quelli che sono stati respinti.
4. Una volta che il programma avrà elaborato i risultati, esso dovrà decidere se avranno superato l'esame più di 8 studenti.

Procediamo con il processo top down per raffinamenti successivi. Cominciamo con una rappresentazione in pseudocodice del top:

Analizzare i risultati dell'esame e decidere se debbano essere aumentate le tasse scolastiche

Ancora una volta, è importante mettere in risalto che il top è una rappresentazione completa del programma, ma che probabilmente saranno necessari diversi raffinamenti, prima che lo pseudocodice possa evolvere naturalmente in un programma C. Il nostro primo raffinamento sarà

Inizializzare le variabili
Prendere in input 10 valutazioni della prova e contare le promozioni e le bocciature
Visualizzare un sommario dei risultati dell'esame e decidere se le tasse scolastiche debbano essere aumentate

Anche a questo punto, sebbene abbiamo una rappresentazione completa dell'intero programma, sarà necessario un successivo raffinamento. Ci impegneremo ora a specificare le variabili. Saranno necessari dei contatori per registrare le promozioni e le bocciature, un contatore sarà utilizzato per controllare l'elaborazione del ciclo e sarà necessaria una variabile per immagazzinare l'input dell'utente. L'istruzione in pseudocodice

Inizializzare le variabili

potrà essere raffinata come segue:

Inizializzare le promozioni a zero
Inizializzare le bocciature a zero
Inizializzare gli studenti a uno

Osservate che sono stati inizializzati soltanto i contatori e i totali. L'istruzione in pseudocodice

Prendere in input 10 valutazioni della prova e contare le promozioni e le bocciature

richiederà un ciclo che prenderà in input consecutivamente i risultati di ognuno degli esami. In questo caso, si conosce in anticipo che ci saranno precisamente dieci risultati di esame, perciò è appropriato utilizzare un ciclo controllato da un contatore. Un comando di selezione doppia all'interno del ciclo, ovverosia *nidificata* nel corpo della iterazione, determinerà se ogni risultato di esame sarà stato una promozione o una bocciatura e incrementerà di conseguenza i contatori appropriati. Il raffinamento della precedente istruzione in pseudocodice sarà quindi

```
Finché il contatore degli studenti è inferiore o uguale a dieci
    Prendere in input il prossimo risultato d'esame

    Se lo studente è stato promosso
        Aggiungere uno ai promossi
    altrimenti
        Aggiungere uno ai bocciati

    Aggiungere uno al contatore degli studenti
```

Osservate l'utilizzo delle righe bianche, per evidenziare la struttura di controllo *Se...altrimenti* e migliorare la leggibilità del programma. L'istruzione in pseudocodice

Visualizzare un sommario dei risultati dell'esame e decidere se le tasse scolastiche debbano essere aumentate

potrà essere raffinata come segue:

```
Visualizzare il numero delle promozioni
Visualizzare il numero delle bocciature
Se più di otto studenti sono stati promossi
    Visualizzare il messaggio "Raise tuition" (aumentare le tasse)
```

Nella Figura 3.9 appare il secondo raffinamento completo. Osservate che le righe vuote sono usate anche per evidenziare il comando *finché* e migliorare la leggibilità del programma.

Ora questo pseudocodice è sufficientemente raffinato per essere convertito in C. Nella Figura 3.10, sono mostrati il programma C e due esempi di esecuzione. Osservate che abbiamo approfittato di una caratteristica del C che consente di incorporare le inizializzazioni nelle dichiarazioni. Una siffatta inizializzazione sarà eseguita durante la compilazione.

Obiettivo efficienza 3.1

Inizializzare le variabili contestualmente alla loro dichiarazione può aiutare a ridurre il tempo di esecuzione di un programma.

Inizializzare le promozioni a zero
Inizializzare le bocciature a zero
Inizializzare gli studenti a uno

Finché il contatore degli studenti è inferiore o uguale a dieci
    Prendere in input il prossimo risultato di esame

    Se lo studente è stato promosso
        Aggiungere uno ai promossi
    altrimenti
        Aggiungere uno ai bocciati

    Aggiungere uno al contatore degli studenti

Visualizzare il numero delle promozioni
Visualizzare il numero delle bocciature
Se più di otto studenti sono stati promossi
    Visualizzare il messaggio "Raise tuition" (aumentare le tasse)

Figura 3.9 Lo pseudocodice per il problema dei risultati dell'esame

Obiettivo efficienza 3.2

Molti dei suggerimenti per migliorare l'efficienza che citiamo in questo libro danno come risultato dei miglioramenti trascurabili e quindi il lettore può essere tentato di ignorarli. Si noti però che l'effetto complessivo di tutti questi miglioramenti dell'efficienza può comportare che un programma venga eseguito molto più velocemente. Inoltre, si realizza un miglioramento significativo quando un suggerimento apparentemente marginale viene applicato in un ciclo che può ripetersi un gran numero di volte.

Ingegneria del software 3.6

L'esperienza ha dimostrato che, su un computer, la parte più difficile della risoluzione di un problema è lo sviluppo dell'algoritmo per la sua soluzione. Normalmente, una volta che sia stato specificato un algoritmo corretto, il processo di produzione di un programma C funzionante sarà lineare.

Ingegneria del software 3.7

Molti programmatori scrivono i programmi senza mai utilizzare uno strumento di sviluppo come lo pseudocodice. Credono che il loro obiettivo principale sia quello di risolvere il problema su un computer, e che scrivere lo pseudocodice serva solo a ritardare la produzione dei risultati finali.

```
1  /* Fig. 3.10: fig03_10.c
2     Analisi dei risultati dell'esame */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     /* inizializza le variabili nelle dichiarazioni */
9     int passes = 0; /* numero di promozioni */
10    int failures = 0; /* numero di bocciature */
11    int student = 1; /* contatore degli studenti */
12    int result; /* risultato di un esame */
13
14    /* elabora 10 studenti; ciclo controllato da un contatore */
15    while ( student <= 10 ) {
16
17       /* sollecita l'utente a inserire i dati e ottiene i valori
            da quest'ultimo */
18       printf( "Enter result (1=pass,2=fail): " );
19       scanf( "%d", &result );
20
21       /* se il risultato è 1, incrementa il numero di promozioni */
22       if ( result == 1 ) {
23          passes = passes + 1;
24       } /* fine del ramo if */
25       else { /* altrimenti incrementa le bocciature */
26          failures = failures + 1;
27       } /* fine del ramo else */
28
29       student = student + 1; /* incrementa il contatore degli studenti */
30    } /* fine del comando while */
31
32    /* fase di terminazione; visualizza il numero di promozioni
         e di bocciature */
33    printf( "Passed %d\n", passes );
34    printf( "Failed %d\n", failures );
35
36    /* se più di otto studenti sono stati promossi, visualizza
         "raise tuition" */
37    if ( passes > 8 ) {
38       printf( "Raise tuition\n" );
39    } /* fine del comando if */
40
41    return 0;   /* chiusura con successo */
42
43 } /* fine della funzione main */
```

Figura 3.10 Il programma C e le esecuzioni di esempio per il problema dei risultati dell'esame (continua)

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Figura 3.10 Il programma C e le esecuzioni di esempio per il problema dei risultati dell'esame

3.11 Gli operatori di assegnamento

Il C fornisce diversi operatori di assegnamento per abbreviare le relative espressioni. Per esempio, l'istruzione

```
c = c + 3;
```

potrà essere abbreviata con l'*operatore di assegnamento addizione* += come in

```
c += 3;
```

L'operatore += aggiunge il valore dell'espressione alla destra dell'operatore alla variabile alla sua sinistra, immagazzinando il risultato in quest'ultima. Ogni istruzione della forma

variabile = *variabile operatore espressione;*

dove *operatore* sia uno degli operatori binari +, -, *, / o % (o altri di cui discuteremo nel Capitolo 10), potrà essere scritta nella forma

variabile operatore= *espressione;*

Di conseguenza l'assegnamento `c += 3` aggiungerà 3 a c. La Figura 3.11 mostra gli operatori aritmetici di assegnamento, alcune espressioni di esempio che li utilizzano e le relative spiegazioni.

Operatore di assegnamento	Espressione di esempio	Spiegazione	Assegna
Assumete: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10 a c`
`-=`	`d -= 4`	`d = d - 4`	`1 a d`
`*=`	`e *= 5`	`e = e * 5`	`20 a e`
`/=`	`f /= 3`	`f = f / 3`	`2 a f`
`%=`	`g %= 9`	`g = g % 9`	`3 a g`

Figura 3.11 Gli operatori aritmetici di assegnamento

3.12 Gli operatori di incremento e di decremento

Il C fornisce anche l'*operatore di incremento* ++ unario e l'*operatore di decremento* -- unario che la Figura 3.12 riassume. Qualora una variabile `c` debba essere incrementata di 1, potrà essere utilizzato l'operatore di incremento ++ invece dell'espressione `c = c + 1` o `c += 1`. Quando gli operatori di incremento o di decremento sono sistemati prima di una variabile, si dicono rispettivamente *operatori di preincremento* o *di predecremento*. Quando gli operatori di incremento o di decremento sono sistemati dopo una variabile, si dicono rispettivamente *operatori di postincremento* o *di postdecremento*. Il preincremento (o il predecremento) di una variabile provocherà in primo luogo l'incremento (o il decremento) di una unità della variabile e, in seguito, il nuovo valore della variabile sarà utilizzato nell'espressione in cui appare. Il postincremento (o il postdecremento) della variabile farà in modo che, nella espressione in cui appare, sia utilizzato il valore corrente della variabile e solo in seguito quest'ultimo sarà incrementato (o decrementato) di una unità.

Operatore	Espressione di esempio	Spiegazione
`++`	`++a`	Incrementa `a` di 1 e quindi utilizza il nuovo valore di `a` nell'espressione in cui essa occorre.
`++`	`a++`	Utilizza il valore corrente di `a` nell'espressione in cui essa occorre e quindi incrementa `a` di 1.
`--`	`--b`	Decrementa `b` di 1 e quindi utilizza il nuovo valore di `b` nell'espressione in cui essa occorre.
`--`	`b--`	Utilizza il valore corrente di `b` nell'espressione in cui essa occorre e quindi decrementa `b` di 1.

Figura 3.12 Gli operatori di incremento e di decremento

Il programma della Figura 3.13 mostra la differenza tra la versione di preincremento e quella di postincremento dell'operatore ++. Il postincremento della variabile `c` provocherà il suo incremento, dopo che la stessa sarà stata utilizzata nella istruzione `printf`. Il preincremento della variabile `c` provocherà il suo incremento, prima che sia utilizzata nella istruzione `printf`.

Il programma visualizzerà il valore di c prima e dopo l'utilizzo dell'operatore ++. L'operatore di decremento (--) funziona in modo simile.

```
1  /* Fig. 3.13: fig03_13.c
2     Preincrementare e postincrementare */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int c;   /* dichiara una variabile */
9
10    /* dimostra l'uso del postincremento */
11    c = 5;                    /* assegna 5 a c */
12    printf( "%d\n", c );      /* visualizza 5 */
13    printf( "%d\n", c++ );    /* visualizza 5 e poi esegue
                                   un postincremento */
14    printf( "%d\n\n", c );    /* visualizza 6 */
15
16    /* dimostra l'uso del preincremento */
17    c = 5;                    /* assegna 5 a c */
18    printf( "%d\n", c );      /* visualizza 5 */
19    printf( "%d\n", ++c );    /* esegue un preincremento
                                   e poi visualizza 6 */
20    printf( "%d\n", c );       /* visualizza 6 */
21
22    return 0;   /* chiusura con successo */
23
24 } /* fine della funzione main */
```

```
5
5
6
5
6
6
```

Figura 3.13 Mostrare la differenza tra il preincremento e il postincremento

Buona abitudine 3.7

Gli operatori unari devono essere sistemati direttamente a ridosso dei loro operandi, senza l'intromissione di spazi.

Le tre istruzioni di assegnamento della Figura 3.10

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

potranno essere scritte in modo più conciso utilizzando gli operatori di assegnamento, come in

```
passes += 1;
failures += 1;
student += 1;
```

o con gli operatori di preincremento, come in

```
++passes;
++failures;
++student;
```

o con gli operatori di postincremento, come in

```
passes++;
failures++;
student++;
```

È importante notare in questo caso che, quando si incrementerà o si decrementerà una variabile in una istruzione isolata, la forma di preincremento e quella di postincremento avranno lo stesso effetto. Il preincremento e il postincremento avranno effetti differenti (e, in modo simile, anche il predecremento e il postdecremento), solo quando una variabile apparirà in una espressione più grande.

Come operando di un operatore di incremento o di decremento, potrà essere utilizzato solo un semplice nome di variabile.

Errore tipico 3.10

Tentare di utilizzare l'operatore di incremento o di decremento in una espressione diversa da un semplice nome di variabile, in altre parole, scrivere `++(x + 1)` *è un errore di sintassi.*

Collaudo e messa a punto 3.4

Il C generalmente non specifica l'ordine in cui saranno valutati gli operandi di un operatore (anche se nel Capitolo 4 vedremo delle eccezioni a questa norma, per alcuni operatori). Di conseguenza, il programmatore dovrà evitare di utilizzare delle istruzioni, con operatori di incremento o di decremento, nelle quali una particolare variabile, che sia stata incrementata o decrementata, compaia più volte.

Operatori	Associatività	Tipo
`++ -- + - (tipo)`	da destra a sinistra	unari
`* / %`	da sinistra a destra	moltiplicativi
`+ -`	da sinistra a destra	additivi
`< <= > >=`	da sinistra a destra	relazionali
`== !=`	da sinistra a destra	di uguaglianza
`?:`	da destra a sinistra	condizionale
`= += -= *= /= %=`	da destra a sinistra	di assegnamento

Figura 3.14 Priorità degli operatori incontrati sinora nel libro

La tabella in Figura 3.14 mostra la priorità e l'associatività degli operatori introdotti sino a questo punto. Gli operatori sono mostrati dall'alto in basso in ordine decrescente di priorità. La seconda colonna descrive l'associatività degli operatori a ogni livello di priorità. Osservate che l'operatore condizionale (`?:`), quelli unari di incremento (`++`), decremento (`--`), addizione (`+`), sottrazione (`-`) e di conversione, nonché gli operatori di assegnamento `=`, `+=`, `-=`, `*=`, `/=` e `%=` associano da destra a sinistra. La terza colonna assegna un nome ai vari gruppi di operatori. Tutti gli altri operatori presenti nella Figura 3.14 associano da sinistra a destra.

Esercizi di autovalutazione

3.1 Completate gli spazi bianchi in ognuna delle seguenti domande.

a) Una procedura per risolvere un problema, definita in termini di azioni che dovranno essere eseguite e dell'ordine in cui quelle azioni dovranno essere eseguite è detta un _____.

b) L'indicazione dell'ordine di esecuzione delle istruzioni da parte del computer è detta _____.

c) Tutti i programmi possono essere scritti in termini di tre tipi di comandi di controllo: _____, _____ e _____.

d) Il comando di selezione _____ è utilizzato per eseguire una azione, qualora una condizione sia vera, e un'altra azione qualora quella condizione sia falsa.

e) Molte istruzioni raggruppate all'interno di parentesi graffe ({ e }) formano un _____.

f) Il comando di iterazione _____ specifica che una istruzione o un loro gruppo dovrà essere eseguito ripetutamente, finché una certa condizione resterà vera.

g) L'iterazione di un insieme di istruzioni per un numero specificato di volte è detta iterazione _____.

h) Quando non è noto in anticipo il numero di volte che un insieme di istruzioni sarà ripetuto, potrà essere utilizzato un valore _____ per terminare l'iterazione.

3.2 Scrivete quattro differenti istruzioni in C che aggiungano 1 alla variabile intera `x`.

3.3 Scrivete una singola istruzione in C per eseguire ognuna delle attività seguenti:

a) Assegnate la somma di `x` e di `y` a `z` e incrementate il valore di `x` di 1 dopo il calcolo.

b) Moltiplicate la variabile `product` per 2, utilizzando l'operatore `*=`.

c) Moltiplicate la variabile `product` per 2, utilizzando gli operatori `=` e `*`.

d) Controllate se il valore della variabile `count` è maggiore di 10. Nel caso lo fosse, visualizzate `"Count is greater than 10"`.

e) Decrementate la variabile `x` di 1 e quindi sottraetela dalla variabile `total`.

f) Aggiungete la variabile `x` alla variabile `total` e quindi decrementate `x` di 1.

g) Calcolate il resto della divisione di `q` per `divisor` e assegnate il risultato a `q`. Scrivete questa istruzione in due modi diversi.

h) Visualizzate il valore `123.4567` con `2` cifre di precisione. Quale valore sarà visualizzato?

i) Visualizzate il valore in virgola mobile `3.14159` con tre cifre a destra del punto decimale. Quale valore sarà visualizzato?

3.4 Scrivete una istruzione C per eseguire ognuna delle seguenti attività.

a) Dichiarate le variabili `sum` e `x` di tipo `int`.

b) Inizializzate la variabile `x` con `1`.

c) Inizializzate la variabile `sum` con `0`.

d) Aggiungete la variabile `x` a `sum` e assegnate il risultato a quest'ultima.

e) Visualizzate `"The sum is: "` seguito dal valore della variabile `sum`.

3.5 Combinate le istruzioni che avete scritto nell'Esercizio 3.4 in un programma che calcoli la somma degli interi da 1 a 10. Utilizzate il comando `while` per iterare le istruzioni di calcolo e di incremento. Il ciclo dovrà terminare quando il valore di `x` diventerà uguale a 11.

3.6 Determinate i valori delle variabili `product` e `x` dopo l'esecuzione del calcolo seguente. Supponete che sia `product` che `x` abbiano il valore 5, quando ogni istruzione comincerà la propria esecuzione.

```
product *= x++;
```

3.7 Scrivete una singola istruzione C che

a) Prenda in input la variabile intera `x` con `scanf`.
b) Prenda in input la variabile intera `y` con `scanf`.
c) Inizializzi la variabile intera `i` con `1`.
d) Inizializzi la variabile intera `power` con `1`.
e) Moltiplichi la variabile `power` per `x` e assegni il risultato a `power`.
f) Incrementi la variabile `y` di `1`.
g) Controlli `i` per verificare se sia inferiore o uguale a `y` nella condizione di un comando `while`.
h) Visualizzi la variabile intera `power` con `printf`.

3.8 Scrivete un programma C che utilizzi le istruzioni dell'Esercizio 3.7 per calcolare `x` elevata alla potenza `y`. Il programma dovrà contenere un comando di controllo di iterazione `while`.

3.9 Identificate e correggete gli errori in ognuna delle seguenti istruzioni:

a)
```
while ( c <= 5 ) {
   product *= c;
   ++c;
```
b)
```
scanf( "%.4f", &value );
```
c)
```
if ( gender == 1 )
   printf( "Woman\n" );
else;
   printf( "Man\n" );
```

3.10 Dite cosa non va nel seguente comando di iterazione `while` (assumete che `z` abbia il valore 100), che si suppone calcoli la somma degli interi partendo da 100 e procedendo a ritroso fino a 1:

```
while (z >= 0)
   sum += z;
```

Risposte agli esercizi di autovalutazione

3.1 a) Algoritmo. b) controllo di programma. c) Sequenza, selezione, iterazione. d) `if...else`. e) Istruzione composta. f) `while`. g) controllata da un contatore. h) Sentinella.

3.2
```
x = x + 1;
x += 1;
++x;
x++;
```

3.3
a)
```
z = x++ + y;
```
b)
```
product *= 2;
```
c)
```
product = product * 2;
```
d)
```
if ( count > 10 )
   printf( "Count is greater than 10.\n" );
```
e)
```
total -= --x;
```

f) `total += x--;`
g) `q %= divisor;`
`q = q % divisor;`
h) `printf( "%.2f", 123.4567 );`
è visualizzato 123,46.
i) `printf( "%.3f\n", 3.14159 );`
è visualizzato 3,142.

3.4 a) `int sum, x;`
b) `x = 1;`
c) `sum = 0;`
d) `sum += x;` o `sum = sum + x;`
e) `printf( "The sum is: %d\n", sum );`

3.5 Si guardi il programma seguente.

```
1  /* Calcola la somma degli interi da 1 a 10 */
2  #include <stdio.h>
3
4  int main()
5  {
6     int sum, x; /* dichiara le variabili sum e x */
7
8     x = 1; /* inizializza x */
9     sum = 0; /* inizializza sum */
10
11    while ( x <= 10 ) { /* itera finché x è minore o uguale a 10 */
12       sum += x; /* aggiunge x a sum */
13       ++x; /* incrementa x */
14    } /* fine del comando while */
15
16    printf( "The sum is: %d\n", sum ); /* visualizza sum */
17
18    return 0;
19 } /* fine della funzione main */
```

3.6 `product = 25, x = 6;`

3.7 a) `scanf( "%d", &x );`
b) `scanf( "%d", &y );`
c) `i = 1;`
d) `power = 1;`
e) `power *= x;`
f) `y++;`
g) `if ( y <= x )`
h) `printf("%d", power);`

3.8 Si guardi il programma seguente.

```
1  /* eleva x alla potenza y */
2  #include <stdio.h>
3
4  int main()
5  {
6     int x,y,i,power; /* dichiara le variabili */
7
8     i = 1; /* inizializza i */
```

```
9       power = 1; /* inizializza power */
10      scanf( "%d", &x ); /* legge il valore di x inserito dall'utente */
11      scanf( "%d", &y ); /* legge il valore di y inserito dall'utente */
12
13      while ( i <= y ) { /* itera finché i è minore o uguale a y */
14         power *= x; /* moltiplica power per x */
15         ++i; /* incrementa i */
16      } /* fine del comando while */
17
18      printf( "%d", power ); /* visualizza power */
19
20      return 0;
21   } /* fine della funzione main */
```

3.9 a) Errore: manca la parentesi graffa di chiusura per il corpo della `while`.
Correzione: aggiungete la parentesi graffa di chiusura dopo l'istruzione `++c;`.

b) Errore: è stata utilizzata la precisione nella specifica di conversione di una `scanf`.
Correzione: rimuovete il `.4` dalla specifica di conversione.

c) Errore: il punto e virgola dopo la parte `else` del comando `if...else` produrrà un errore logico: la seconda `printf` sarà eseguita sempre.
Correzione: rimuovete il punto e virgola dopo l'`else`.

3.10 Il valore della variabile `z` non sarà mai modificato all'interno del comando `while`. È stato quindi creato un ciclo infinito. Per evitare il ciclo infinito, `z` dovrà essere decrementata in modo da assumere, a un certo momento, il valore `0`.

Esercizi

3.11 Identificate e correggete gli errori in ognuna delle seguenti istruzioni [*Nota*: potrà esserci più di un errore in ognuno dei pezzi di codice]:

a)
```
if ( age >= 65 );
   printf( "Age is greater than or equal to 65\n" );
else
   printf( "Age is less than 65\n" );
```

b)
```
int x = 1, total;

while ( x <= 10 ) {
   total += x;
   ++x;
}
```

c)
```
While ( x <= 100 )
   total += x;
   ++x;
```

d)
```
while ( y > 0 ) {
   printf( "%d\n", y );
   ++y;
}
```

3.12 Riempite gli spazi in ognuna delle frasi seguenti:

a) La soluzione di ogni problema richiede l'esecuzione di una serie di azioni in un _____ specifico.

b) Un sinonimo di procedura è _____.

c) Una variabile che accumula la somma di diversi numeri è un _____.

d) Il processo in cui si impostano certe variabili con dei valori specifici all'inizio di un programma è detto _____.

e) Un valore speciale utilizzato per indicare la "fine della immissione dei dati" è detto valore _____, _____, _____ o _____.
f) Un _____ è una rappresentazione grafica di un algoritmo.
g) In un diagramma di flusso, l'ordine in cui dovranno essere eseguiti i passi sarà indicato dai simboli di _____.
h) Il simbolo di terminazione indica l'_____ e la _____ di ogni algoritmo.
i) Il simbolo rettangolo corrisponde ai calcoli, che normalmente saranno eseguiti dalle istruzioni _____, e alle operazioni di input/output, che normalmente saranno eseguite dalle chiamate alle funzioni _____ e _____ della libreria standard.
j) L'elemento scritto all'interno di un simbolo di decisione è detto _____.

3.13 Che cosa visualizzerà il seguente programma?

```
1  #include <stdio.h>
2
3  int main()
4  {
5     int x = 1, total = 0, y;
6
7     while ( x <= 10 ) {
8     y = x * x;
9        printf( "%d\n", y );
10       total += y;
11       ++x;
12    }
13
14    printf( "Total is %d\n", total );
15
16    return 0;
17 }
```

3.14 Scrivete una singola istruzione in pseudocodice che esegua ognuna delle seguenti azioni:
a) Visualizzate il messaggio «`Enter two numbers`".
b) Assegnate a `p` la somma delle variabili `x`, `y` e `z`.
c) La seguente condizione dovrà essere controllata in un comando di selezione `if...else`: il valore corrente della variabile `m` è maggiore del doppio di quello contenuto nella variabile `v`?
d) Ottenete dalla tastiera dei valori per le variabili `s`, `r` e `t`.

3.15 Formulate un algoritmo in pseudocodice per ognuna delle seguenti attività:
a) Ottenete due numeri dalla tastiera, calcolatene la somma e visualizzate il risultato.
b) Ottenete due numeri dalla tastiera e quindi determinate e visualizzate il maggiore dei due, se c'è.
c) Ottenete una serie di numeri positivi dalla tastiera e quindi determinate e visualizzate la loro somma. Supponete che l'utente immetta il valore sentinella `-1` per indicare la "fine dell'immissione dei dati".

3.16 Stabilite quali delle seguenti affermazioni sono *vere* e quali sono *false*. Qualora una affermazione sia *falsa*, spiegatene il motivo.
a) L'esperienza ha dimostrato che la parte più difficile della risoluzione di un problema, su un computer, è la produzione di un programma C funzionante.
b) Un valore sentinella dovrà essere tale che non possa essere confuso con un valore legittimo per i dati.
c) Le linee di flusso indicano le azioni da eseguire.

d) Le condizioni scritte all'interno dei simboli di decisione contengono sempre degli operatori aritmetici (ovverosia +, -, *, / e %).
e) Nel processo top down per raffinamenti successivi, ogni raffinamento è una rappresentazione completa dell'algoritmo.

Per gli Esercizi dal 3.17 al 3.21, eseguite ognuno di questi passi:

1. Leggete l'enunciazione del problema.
2. Formulate l'algoritmo usando lo pseudocodice e il processo top down per raffinamenti successivi.
3. Scrivete un programma C.
4. Collaudate, mettete a punto ed eseguite il programma C.

3.17 Gli automobilisti sono interessati al numero di chilometri percorsi dalle proprie automobili. Un automobilista ha mantenuto traccia di diversi pieni di benzina, registrando i chilometri percorsi e i litri utilizzati per ogni pieno. Sviluppate un programma che prenda in input i chilometri percorsi e i litri utilizzati per ogni pieno. Il programma dovrà calcolare e visualizzare i chilometri per litro ottenuti da ogni pieno. Dopo aver elaborato tutte le informazioni in input, il programma dovrà calcolare e visualizzare anche i chilometri per litro ottenuti complessivamente da tutti i pieni. Di seguito è riportato un esempio dell'interazione per quanto riguarda l'input/output dei dati:

```
Enter the gallons used (-1 to end): 12.8
Enter the miles driven: 287
The miles / gallon for this tank was 22.421875

Enter the gallons used (-1 to end): 10.3
Enter the miles driven: 200
The miles / gallon for this tank was 19.417475

Enter the gallons used (-1 to end): 5
Enter the miles driven: 120
The miles / gallon for this tank was 24.000000

Enter the gallons used (-1 to end): -1

The overall average miles/gallon was 21.601423
```

3.18 Sviluppate un programma C, che determini se il cliente di un grande magazzino abbia superato il limite di credito sul suo conto. Per ogni cliente saranno disponibili i seguenti dati:

1. Il numero di conto
2. Il saldo all'inizio del mese
3. Il totale di tutti gli articoli che il cliente ha messo in conto, durante il mese corrente.
4. Il totale di tutti i crediti applicati al conto di questo cliente, durante il mese corrente.
5. Il limite di credito concesso

Il programma dovrà prendere in input tutti questi dati, calcolare il nuovo saldo (= saldo iniziale + articoli messi in conto – crediti) e determinare se il nuovo saldo superi il limite di credito del cliente. Per quei clienti che avranno superato il limite di credito, il programma dovrà visualizzare il loro numero di conto, il limite di credito, il nuovo saldo e il messaggio "Credit limit exceeded". Di seguito è riportato un esempio dell'interazione per quanto riguarda l'input/output dei dati:

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
Account:      100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter account number (-1 to end): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00

Enter account number (-1 to end): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00

Enter account number (-1 to end): -1
```

3.19 Una grande industria chimica retribuisce i propri venditori basandosi sulle provvigioni. Il venditore riceve $ 200 la settimana più il 9% delle proprie vendite lorde portate a termine durante la settimana. Per esempio, un venditore che in una settimana venda prodotti chimici per un valore di $ 5000, riceverà $ 200 più il 9% di $ 5000, ovverosia un totale di $ 650. Scrivete un programma, che prenda in input le vendite lorde di ogni venditore per l'ultima settimana e quindi calcoli e visualizzi il salario per ognuno di loro. Elaborate i conti di un venditore per volta. Di seguito è riportato un esempio dell'interazione per quanto riguarda l'input/output dei dati:

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 1234.56
Salary is: $311.11

Enter sales in dollars (-1 to end): 1088.89
Salary is: $298.00

Enter sales in dollars (-1 to end): -1
```

3.20 L'interesse semplice su un mutuo è calcolato dalla formula

```
interest = principal * rate * days / 365
```

La formula precedente presume che `rate` sia il tasso di interesse annuale e quindi include la divisione per 365 (giorni). Sviluppate un programma che prenda in input `principal` (capitale), `rate` (tasso) e `days` (giorni) per diversi mutui e visualizzi l'interesse semplice per ogni mutuo, utilizzando la formula precedente. Di seguito è riportato un esempio dell'interazione per quanto riguarda l'input/output dei dati:

```
Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.400

Enter loan principal (-1 to end): 10000.00
Enter interest rate: .09
Enter term of the loan in days: 1460
The interest charge is $3600.00

Enter loan principal (-1 to end): -1
```

3.21 Sviluppate un programma che determini la paga lorda per ognuno dei diversi impiegati. L'azienda, per le prime 40 ore lavorate da ogni impiegato, paga il "salario orario di base", mentre per tutte le ore lavorate in aggiunta alle 40, elargisce "una volta e mezza il salario di base". Vi sono stati forniti: una lista degli impiegati dell'azienda, il numero di ore lavorate da ogni impiegato nell'ultima settimana e il salario orario di base di ogni impiegato. Il vostro programma dovrà prendere in input, per ogni impiegato, le suddette informazioni e dovrà quindi determinare e visualizzare la paga lorda di ognuno di loro. Di seguito è riportato un esempio dell'interazione per quanto riguarda l'input/output dei dati:

```
Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1
```

3.22 Scrivete un programma che dimostri la differenza tra il predecremento e il postdecremento, utilizzando l'operatore di decremento --.

3.23 Scrivete un programma che utilizzi un ciclo, per visualizzare i numeri da 1 a 10 a fianco a fianco sulla stessa riga e con tre spazi tra ognuno di essi.

3.24 Il processo di ricerca del numero più grande (ovverosia, del valore massimo in un gruppo di numeri) è utilizzato frequentemente nelle applicazioni per computer. Un programma che determini il vincitore di una gara di vendite, per esempio, prenderà in input il numero degli articoli venduti da ogni venditore. Vincerà la gara il venditore che avrà venduto il maggior numero di articoli. Scrivete un programma in pseudocodice e quindi un programma vero e proprio, che prenda in input una serie di 10 numeri e in seguito determini e visualizzi il maggiore di quelli. [*Suggerimento*: il vostro programma dovrà utilizzare tre variabili, come segue]:

`counter`: un contatore per contare fino a 10 (ovverosia, per mantenere traccia di quanti numeri saranno stati immessi e per determinare quando tutti i 10 numeri saranno stati elaborati).
`number`: il numero corrente immesso nel programma.
`largest`: il numero più grande trovato sino a questo punto.

3.25 Scrivete un programma che utilizzi un ciclo per visualizzare la seguente tabella di valori:

```
N      10*N   100*N  1000*N

1      10     100    1000
2      20     200    2000
3      30     300    3000
4      40     400    4000
5      50     500    5000
6      60     600    6000
7      70     700    7000
8      80     800    8000
9      90     900    9000
10     100    1000   10000
```

La sequenza di escape di tabulazione, `\t`, potrà essere utilizzato nella istruzione `printf` per separare le colonne con delle tabulazioni.

3.26 Scrivete un programma che utilizzi un ciclo per produrre la seguente tabella di valori:

```
A   A+2 A+4 A+6

3   5   7   9
6   8   10  12
9   11  13  15
12  14  16  18
15  17  19  21
```

3.27 Trovate i due numeri maggiori tra 10 valori, usando un approccio simile all'Esercizio 3.24. [*Nota*: potrete prendere in input ogni valore soltanto una volta.]

3.28 Modificate il programma della Figura 3.10, in modo da convalidare l'input. Reiterate su ogni input, qualora il valore immesso fosse diverso da 1 o da 2, finché l'utente non immetta un valore corretto.

3.29 Che cosa visualizzerà il seguente programma?

```
1  #include <stdio.h>
2
3  /* l'esecuzione del programma inizia dalla funzione main */
4  int main()
5  {
6     int count = 1; /* inizializza count */
7
8     while ( count <= 10 ) { /* itera 10 volte */
9
10       /* visualizza una linea di testo */
11       printf( "%s\n", count % 2 ? "****" : "++++++++" );
12       count++; /* incrementa count */
13    } /* fine del comando while */
```

```
14
15      return 0; /* indica che il programma è terminato con successo */
16
17   } /* fine della funzione main */
```

3.30 Che cosa visualizzerà il seguente programma?

```
1   #include <stdio.h>
2
3   /* l'esecuzione del programma inizia dalla funzione main */
4   int main()
5   {
6      int row = 10; /* inizializza row */
7      int column; /* dichiara column */
8
9      while ( row >= 1 ) { /* cicla finché non si ha row < 1 */
10           column = 1; /* imposta column a 1 all'inizio dell'iterazione */
11
12           while ( column <= 10 ) { /* itera 10 volte */
13              printf( "%s", row % 2 ? "<": ">" ); /* visualizzazione */
14              ++column; /* incrementa column */
15           } /* fine del comando while interno */
16
17              --row; /* decrementa row */
18              printf( "\n" ); /* inizia una nuova linea di output */
19        } /* fine del comando while esterno */
20
21        return 0; /* indica che il programma è terminato con successo */
22
23     } /* fine della funzione main */
```

3.31 (*Problema dell'else appeso*) Determinate l'output d'ognuna delle seguenti istruzioni quando `x` è `9` e `y` è `11` e quando `x` è `11` e `y` è `9`. Osservate che il compilatore ignora i rientri all'interno di un programma C. Il compilatore C inoltre assocerà sempre un `else` all'`if` precedente, sempre che non gli sia stato detto di fare altrimenti, attraverso la disposizione delle parentesi graffe `{ }`. Dato che, a prima vista, il programmatore non può essere sicuro della corrispondenza di un `else` con un `if`, questo problema è definito "dell'else appeso". Abbiamo eliminato i rientri dal codice successivo, per rendere più interessante il problema. [*Suggerimento*: applicate le convenzioni dei rientri che avete appreso.]

a)
```
if ( x < 10 )
if ( y > 10 )
printf( "*****\n" );
else
printf( "#####\n" );
printf( "$$$$$\n" );
```

b)
```
if ( x < 10 ) {
if ( y > 10 )
printf( "*****\n" );
}
else {
printf( "#####\n" );
printf( "$$$$$\n" );
}
```

3.32 (*Un altro problema dell'else appeso*) Modificate il codice seguente in modo che produca l'output mostrato. Utilizzate le tecniche di rientro appropriate. Non potrete eseguire altre modifiche che non siano degli inserimenti di parentesi graffe. Il compilatore ignora i rientri in un programma. Abbiamo

eliminato i rientri dal codice seguente, per rendere più interessante il problema. [*Nota*: è probabile che non sia necessaria alcuna modifica.]

```
if ( y == 8 )
if ( x == 5 )
printf( "@@@@@\n" );
else
printf( "#####\n" );
printf( "$$$$$\n" );
printf( "&&&&&\n" );
```

a) Assumendo `x = 5` e `y = 8`, sarà prodotto l'output seguente.

```
@@@@@
$$$$$
&&&&&
```

b) Assumendo `x = 5` e `y = 8`, sarà prodotto l'output seguente.

```
@@@@@
```

c) Assumendo `x = 5` e `y = 8`, sarà prodotto l'output seguente.

```
@@@@@
&&&&&
```

d) Assumendo `x = 5` e `y = 7`, sarà prodotto l'output seguente. [*Nota*: le ultime tre `printf` sono tutte parti di un'istruzione composta.]

```
#####
$$$$$
&&&&&
```

3.33 Scrivete un programma che legga in input il lato di un quadrato e quindi lo disegni utilizzando degli asterischi. Il vostro programma dovrà funzionare con tutti i quadrati con dimensioni dei lati comprese tra 1 e 20. Per esempio, se la dimensione letta dal vostro programma fosse 4, dovrebbe visualizzare:

```
****
****
****
****
```

3.34 Modificate il programma che avete scritto nell'Esercizio 3.33 in modo che visualizzi un quadrato vuoto. Per esempio, se la dimensione letta dal vostro programma fosse 5, dovrebbe visualizzare:

```
*****
*   *
*   *
*   *
*****
```

3.35 Un palindromo è un numero o una frase di testo che, da sinistra a destra o da destra a sinistra, si legge nello stesso modo. Per esempio, ognuno dei seguenti interi di cinque cifre è un palindromo: 12321, 55555, 45554 e 11611. Scrivete un programma che legga in input un intero di cinque cifre e determini se si tratta o no di un palindromo. [*Suggerimento*: per scindere il numero nelle sue singole cifre, utilizzate gli operatori di divisione e di resto.]

3.36 Prendete in input un intero contenente soltanto degli 0 e degli 1, ovverosia un intero "binario", e visualizzate il suo equivalente decimale. [*Suggerimento*: per prelevare le cifre del numero "binario" una per volta da destra a sinistra, utilizzate gli operatori di divisione e di resto. Nel sistema numerico decimale, la cifra più a destra ha un valore posizionale di 1 e quelle che si susseguono a sinistra hanno un valore posizionale di 10, poi 100, poi 1000 ecc.; allo stesso modo, nel sistema numerico binario, la cifra più a destra ha un valore posizionale di 1 e quelle che si susseguono a sinistra hanno un valore posizionale di 2, poi 4, poi 8, ecc. Di conseguenza il numero 234 potrà essere interpretato come 4 * 1 + 3 * 10 + 2 * 100. L'equivalente decimale del binario 1101 sarà 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 o 1 + 0 + 4 + 8 o 13.]

3.37 In che modo potete determinare quanto è realmente veloce il vostro computer? Scrivete un programma con un ciclo `while` che conti uno per volta da 1 a 3.000.000. Ogni volta che il conto raggiungerà un multiplo di 1.000.000, visualizzerete il numero sullo schermo. Utilizzate il vostro orologio per misurare il tempo impiegato da ogni milione di iterazioni all'interno del ciclo.

3.38 Scrivete un programma che visualizzi 100 asterischi uno per volta. Ogni dieci asterischi, il vostro programma dovrà visualizzare un carattere newline. [*Suggerimento*: contate da 1 a 100. Usate l'operatore di resto per individuare tutte le volte che il contatore raggiunge un multiplo di 10.]

3.39 Scrivete un programma che legga in input un intero e, in seguito, determini e visualizzi quanti 7 sono compresi nelle cifre dell'intero.

3.40 Scrivete un programma che visualizzi il seguente disegno di una scacchiera:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Il vostro programma dovrà utilizzare soltanto tre istruzioni `printf`, una per ognuna delle forme seguenti:

```
printf( "* " );
printf( " " );
printf( "\n" );
```

3.41 Scrivete un programma che continui a visualizzare i multipli dell'intero 2, ovverosia 2, 4, 8, 16, 32, 64, ecc. Il vostro ciclo non dovrà mai terminare (dovrete insomma creare un ciclo infinito). Che cosa succederà quando eseguirete questo programma?

3.42 Scrivete un programma che legga il raggio di un cerchio (come un valore di tipo `float`) e quindi calcoli e visualizzi il diametro, la circonferenza e l'area. Utilizzate il valore 3.14159 per π.

3.43 Cosa non va nella seguente istruzione? Riscrivete l'istruzione in modo che esegua quello che probabilmente il programmatore stava tentando di ottenere.

```
printf ( "%d", ++( x + y ) );
```

3.44 Scrivete un programma che legga in input tre valori di tipo `float` diversi da zero e, quindi, determini e visualizzi se possono rappresentare i lati di un triangolo.

3.45 Scrivete un programma che legga in input tre interi diversi da zero e, quindi, determini e visualizzi se possono essere i lati di un triangolo rettangolo.

3.46 Una azienda vuole trasmettere dei dati sulla linea telefonica, ma i suoi responsabili sono preoccupati dal fatto che i propri telefoni potrebbero essere spiati. Tutti i loro dati sono trasmessi come interi di quattro cifre. Vi hanno quindi chiesto di scrivere un programma che crittografi i loro dati in modo che possano essere trasmessi con maggior sicurezza. Il vostro programma dovrà leggere un intero di quattro cifre e crittografarlo nel modo seguente: sostituite ogni cifra con il resto ottenuto dalla divisione di (la somma di quella cifra più 7) per 10. In seguito, scambiate la prima cifra con la terza, e scambiate la seconda cifra con la quarta. Visualizzate quindi l'intero crittografato. Scrivete un programma a parte che prenda in input un intero crittografato di quattro cifre e lo decifri, in modo da formare il numero originale.

3.47 Il fattoriale di un intero non negativo n si scrive $n!$ (pronunciato "fattoriale di n") ed è definito come segue:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (per valori di n maggiori o uguali a 1)

e

$n! = 1$ (per $n = 0$).

Per esempio, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ che è 120.

a) Scrivete un programma che legga in input un intero non negativo e quindi calcoli e visualizzi il suo fattoriale.

b) Scrivete un programma che approssimi il valore della costante matematica e, utilizzando la formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Scrivete un programma che approssimi il valore di e^x, utilizzando la formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Capitolo 4

Il controllo del programma in C

Obiettivi

- Essere in grado di utilizzare i comandi di iterazione `for` e `do/while`.
- Comprendere la selezione multipla, utilizzando il comando di selezione `switch`.
- Essere in grado di utilizzare le istruzioni `break` e `continue` per il controllo del programma.
- Essere in grado di utilizzare gli operatori logici.

4.1 Introduzione

A questo punto, il lettore dovrebbe essere a proprio agio con il processo di scrittura di un semplice ma completo programma C. In questo capitolo, sarà trattata più dettagliatamente l'iterazione e, nello stesso tempo, saranno presentati altri comandi di controllo per quest'ultima, ovverosia il comando `for` e quello `do...while`. Sarà introdotto anche il comando di selezione multipla `switch`. Discuteremo dell'istruzione `break`, per uscire immediatamente e rapidamente da certi comandi di controllo, e dell'istruzione `continue`, per ignorare la parte rimanente del corpo di un comando di iterazione e procedere con la successiva iterazione del ciclo. Il capitolo discuterà inoltre degli operatori logici utilizzati per combinare fra loro le condizioni e, infine, concluderà con un riassunto dei principi della programmazione strutturata presentati nel Capitolo 3 e nel Capitolo 4.

4.2 Gli elementi dell'iterazione

La maggior parte dei programmi richiede delle iterazioni o *cicli*. Un *ciclo (loop)* è un gruppo di istruzioni che il computer eseguirà ripetutamente, finché una certa *condizione di continuazione del ciclo* rimarrà vera. Sinora abbiamo discusso di due tipi di iterazione:

1. L'iterazione controllata da un contatore
2. L'iterazione controllata da un valore sentinella.

L'iterazione controllata da un contatore è detta a volte *iterazione definita*, perché conosciamo esattamente in anticipo il numero di volte che il ciclo sarà eseguito. L'iterazione controllata da un valore sentinella è detta a volte *iterazione indefinita*, perché non è noto in anticipo il numero di volte che il ciclo sarà eseguito.

Nella iterazione controllata da un contatore, è utilizzata una *variabile di controllo* per contare il numero delle iterazioni. La variabile di controllo sarà incrementata (di solito di 1) ogni volta che sarà eseguito il gruppo di istruzioni. Il ciclo terminerà quando il valore della

variabile di controllo indicherà che sarà stato eseguito il numero corretto di iterazioni; a quel punto, il computer potrà continuare l'esecuzione con l'istruzione successiva alla struttura di controllo.

I valori sentinella sono utilizzati per controllare una iterazione quando:

1. il numero preciso delle iterazioni non è noto in anticipo e
2. il ciclo include delle istruzioni che otterranno dei dati ogni volta che quest'ultimo sarà eseguito.

Il valore sentinella indica la "fine dei dati" e sarà immesso dall'utente dopo che tutti i veri elementi informativi saranno stati forniti al programma. I valori sentinella dovranno essere distinti dalle informazioni vere e proprie.

4.3 Iterazione controllata da un contatore

Una iterazione controllata da un contatore richiede:

1. Il *nome* di una variabile di controllo (o contatore del ciclo).
2. Il *valore iniziale* della variabile di controllo.
3. L'*incremento (o decremento)* con cui la variabile di controllo sarà modificata ogni volta nel corso del ciclo.
4. La condizione che verificherà il *valore finale* della variabile di controllo (ovverosia, quella che determinerà se il ciclo dovrà continuare).

Considerate il semplice programma mostrato nella Figura 4.1 che visualizza i numeri da 1 a 10. La dichiarazione

```
int counter = 1; /* inizializzazione */
```

assegna un nome alla variabile di controllo (`counter`), la dichiara di tipo intero, le riserva uno spazio nella memoria e le assegna il *valore iniziale* `1`. Questa dichiarazione non è un'istruzione eseguibile.

Avremmo anche potuto dichiarare e inizializzare `counter` con le istruzioni

```
int counter;
counter = 1;
```

La dichiarazione, anche in questo caso, non è una istruzione eseguibile, mentre l'assegnamento lo è. In seguito utilizzeremo entrambi i metodi per l'inizializzazione delle variabili.

L'istruzione

```
++counter; /* incremento */
```

incrementerà il contatore del ciclo di 1 ogni volta che questo sarà eseguito. La condizione di continuazione del ciclo nel comando `while` controllerà se il valore della variabile di controllo sia inferiore o uguale a `10`, ovverosia l'ultimo valore per il quale la condizione risulterà vera.

Osservate che il corpo di questo `while` sarà eseguito anche quando la variabile di controllo varrà `10`. Il ciclo terminerà quando la variabile di controllo avrà superato `10` (`counter` avrà assunto il valore `11`).

```
1  /* Fig. 4.1: fig04_01.c
2     Iterazione controllata da un contatore */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int counter = 1; /* inizializzazione */
9
10    while ( counter <= 10 ) { /* condizione di iterazione */
11       printf ( "%d\n", counter ); /* visualizza il contatore */
12       ++counter; /* incremento */
13    } /* fine del comando while */
14
15    return 0; /* indica che il programma è terminato con successo */
16
17 } /* fine della funzione main */
```

```
1
2
3
4
5
6
7
8
9
10
```

Figura 4.1 Iterazione controllata da un contatore

Normalmente, i programmatori C renderebbero più conciso il programma della Figura 4.1, inizializzando `counter` a `0` e sostituendo il comando `while` con

```
while ( ++counter <= 10 )
   printf ( "%d\n", counter );
```

Questo codice consentirebbe di risparmiare una istruzione, poiché l'incremento sarebbe eseguito direttamente nella condizione del ciclo `while`, prima che la stessa sia verificata. Questo codice eliminerebbe anche le parentesi graffe intorno al corpo del `while`, perché a quel punto esso conterrebbe una sola istruzione. Scrivere il codice in un modo così conciso richiede una certa pratica.

Errore tipico 4.1

Dato che i valori in virgola mobile potrebbero essere approssimati, utilizzare delle variabili di quel tipo per il controllo dei cicli, potrebbe produrre dei valori imprecisi del contatore e provocare dei controlli di terminazione non accurati.

Collaudo e messa a punto 4.1

Controllate le iterazioni determinate con dei valori interi.

Buona abitudine 4.1

Fate rientrare con degli spazi e delle tabulazioni le istruzioni nel corpo di ognuno dei comandi di controllo.

Buona abitudine 4.2

Inserite una riga vuota prima e dopo ogni principale comando di controllo in modo da evidenziarlo all'interno del programma.

Buona abitudine 4.3

Troppi livelli di nidificazione potrebbero rendere il programma incomprensibile. Come regola generale, cercate di evitare l'utilizzo di più di tre livelli di nidificazione.

Buona abitudine 4.4

Combinando le spaziature verticali inserite prima e dopo i comandi di controllo con il rientro dei loro corpi in una posizione più interna rispetto alle relative testate, darete al vostro programma una apparenza bidimensionale che ne migliorerà enormemente la leggibilità.

4.4 Il comando di iterazione for

Il comando di iterazione `for` gestisce automaticamente tutti i dettagli di una iterazione controllata da un contatore. Riscriviamo il programma della Figura 4.1, per illustrare la potenza della istruzione `for`. Il risultato sarà mostrato nella Figura 4.2.

```
1  /* Fig. 4.2: fig04_02.c
2     Iterazione controllata da un contatore con il comando for */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int counter; /* dichiara un contatore */
9
10    /* inizializzazione, condizione di iterazione e incremento
11       sono tutti inclusi nell'intestazione del comando for */
12    for ( counter = 1; counter <= 10; counter++ ) {
13       printf( "%d\n", counter );
14    } /* fine del comando for */
15
16    return 0; /* indica che il programma è terminato con successo */
17
18 } /* fine della funzione main */
```

Figura 4.2 Iterazione controllata da un contatore con il comando for

Il programma opererà come segue. Nel momento in cui il comando `for` incomincerà a essere eseguito, la variabile di controllo `counter` sarà inizializzata a `1`. In seguito sarà verifica-

ta la condizione di continuazione del ciclo: `counter <= 10`. Dato che il valore iniziale di `counter` sarà `1`, la condizione sarà soddisfatta e quindi l'istruzione `printf` (riga 13) visualizzerà il valore di `counter`, vale a dire `1`. La variabile di controllo sarà quindi incrementata dalla espressione `counter++` e il ciclo ripartirà nuovamente, con la verifica della propria condizione di continuazione. Dato che la variabile di controllo sarà ora uguale a `2`, il valore finale non sarà stato ancora superato e di conseguenza il programma eseguirà ancora l'istruzione `printf`. Questo processo continuerà finché la variabile di controllo `counter` sarà stata incrementata al suo valore finale `11`. Ciò farà fallire la verifica della condizione di continuazione per il ciclo e ne provocherà la terminazione. Il programma continuerà eseguendo la prima istruzione dopo il comando `for` (in questo caso, l'istruzione `return` alla fine dello stesso).

La Figura 4.3 illustra più dettagliatamente il comando `for` della Figura 4.2. Notate che il comando `for` "fa tutto": specifica ogni elemento necessario a una iterazione controllata da un contatore con una variabile di controllo. Qualora nel corpo del `for` siano presenti più istruzioni, saranno necessarie delle parentesi graffe per definire il corpo del ciclo.

Osservate che la Figura 4.2 utilizza la condizione di continuazione del ciclo `counter <= 10`. Nel caso in cui il programmatore avesse erroneamente scritto `counter < 10`, il ciclo sarebbe stato eseguito soltanto 9 volte. Questo sarebbe stato un tipico errore logico chiamato *"errore di imprecisione di uno"*.

Errore tipico 4.2

Utilizzare un operatore relazionale errato o usare un valore finale errato per il contatore del ciclo, all'interno della condizione di un comando `while` *o* `for`*, potrà provocare degli errori di imprecisione di uno.*

Collaudo e messa a punto 4.2

Usare il valore finale insieme all'operatore relazionale `<=`*, all'interno della condizione di un comando* `while` *o* `for`*, aiuterà a evitare gli errori di imprecisione di uno. In un ciclo utilizzato per visualizzare i valori da 1 a 10, per esempio, la condizione di continuazione del ciclo dovrà essere* `counter <= 10` *piuttosto che* `counter < 11` *o* `counter < 10`.

Il formato generale del comando `for` è

```
for ( espressione1; espressione2; espressione3 )
istruzione
```

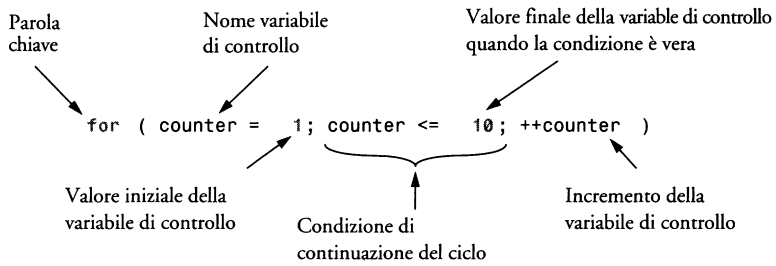


Figura 4.3 I componenti dell'intestazione del for

dove *espressione1* inizializzerà la variabile di controllo del ciclo, *espressione2* sarà la condizione di continuazione ed *espressione3* incrementerà la variabile di controllo. Nella maggior parte dei casi il comando `for` potrà essere rappresentato da un `while` equivalente, come segue:

```
espressione1;
while ( espressione2 ) {
   istruzione
   espressione3;
}
```

C'è una eccezione a questa regola che sarà discussa nella Sezione 4.9.

Spesso, *espressione1* ed *espressione3* saranno delle liste di espressioni separate da virgole. Utilizzate in questo modo, le virgole saranno in realtà degli *operatori virgola* che garantiranno la valutazione da sinistra a destra dell'elenco di espressioni. Il valore e il tipo di una lista di espressioni separate da virgole saranno quelli dell'ultima espressione della lista. L'operatore virgola è utilizzato molto spesso nei comandi `for`. Il suo utilizzo principale è quello di consentire al programmatore di utilizzare le inizializzazioni e/o le espressioni di incremento multiple. Per esempio, in un singolo comando `for` potranno esserci due variabili di controllo che dovranno essere inizializzate e incrementate.

Ingegneria del software 4.1

Nelle sezioni di inizializzazione e di incremento di un comando **`for`**, *inserite solo delle espressioni che coinvolgano le variabili di controllo. La gestione delle altre variabili dovrebbe comparire o prima del ciclo (se dovranno essere eseguite una sola volta, come le istruzioni di inizializzazione), oppure all'interno del corpo del ciclo (se dovranno essere eseguite una volta per ogni iterazione, come le istruzioni di incremento o di decremento).*

Le tre espressioni di un comando `for` sono opzionali. Qualora *espressione2* sia omessa, il C presumerà che la condizione sia sempre vera, creando quindi un ciclo infinito. Si potrebbe omettere *espressione1*, qualora la variabile di controllo fosse stata inizializzata altrove nel programma. L'*espressione3* potrà essere omessa, qualora l'incremento fosse calcolato dalle istruzioni interne al corpo del comando `for`, o qualora non sia richiesto alcun incremento. L'espressione di incremento all'interno del comando `for` agirà come se fosse una istruzione C a sé stante, posta alla fine del corpo del `for`. Di conseguenza, nella sezione di incremento del comando `for`, le espressioni

```
counter = counter + 1
counter += 1
++counter
counter++
```

saranno tutte equivalenti. Molti programmatori C preferiscono la forma `counter++`, perché l'incremento avverrà dopo che sarà stato eseguito il ciclo all'interno del corpo. Per questo motivo la forma del postincremento sembra più naturale. Dato che il preincremento o il postincremento della variabile in questo caso non appare in una espressione, entrambe le forme di incremento avranno lo stesso effetto. Entrambi i punti e virgola nel comando del `for` sono obbligatori.

Errore tipico 4.3

Usare le virgole invece dei punti e virgola nell'intestazione di un `for` *è un errore di sintassi.*

Errore tipico 4.4

L'inserimento di un punto e virgola immediatamente alla destra della intestazione di un `for` *renderà il corpo di quel comando* `for` *un'istruzione vuota. Questo è normalmente un errore di logica.*

4.5 Il comando for: note e osservazioni

1. L'inizializzazione, la condizione di continuazione del ciclo e l'incremento potranno contenere delle espressioni aritmetiche. Per esempio, supponete che `x = 2` e `y = 10`, l'istruzione

   ```
   for ( j = x; j <= 4 * x * y; j += y / x )
   ```

   sarà equivalente alla istruzione

   ```
   for ( j = 2; j <= 80; j += 5 )
   ```

2. "L'incremento" potrà essere negativo (in tal caso sarà in realtà un decremento e il ciclo effettivamente conterà a ritroso).
3. Qualora la condizione di continuazione del ciclo sia falsa sin dall'inizio, la porzione all'interno del corpo del ciclo non sarà eseguita. L'esecuzione procederà invece con l'istruzione successiva al comando `for`.
4. La variabile di controllo è spesso visualizzata o utilizzata nei calcoli all'interno del corpo di un ciclo, ma non è necessario che lo sia. È tipico usare la variabile di controllo per controllare l'iterazione, senza menzionarla mai nel corpo del ciclo.
5. Il diagramma del comando `for` è molto simile a quello del comando `while`. Per esempio, il diagramma di flusso dell'istruzione `for`

   ```
   for ( counter = 1; counter <= 10; counter++ )
      printf("%d", counter);
   ```

è mostrato nella Figura 4.4. Questo diagramma di flusso rende evidente che l'inizializzazione sarà effettuata solo una volta, mentre l'incremento sarà eseguito solo dopo che sarà stata completata l'istruzione all'interno del corpo. Osservate che (a parte i cerchietti e le frecce) il

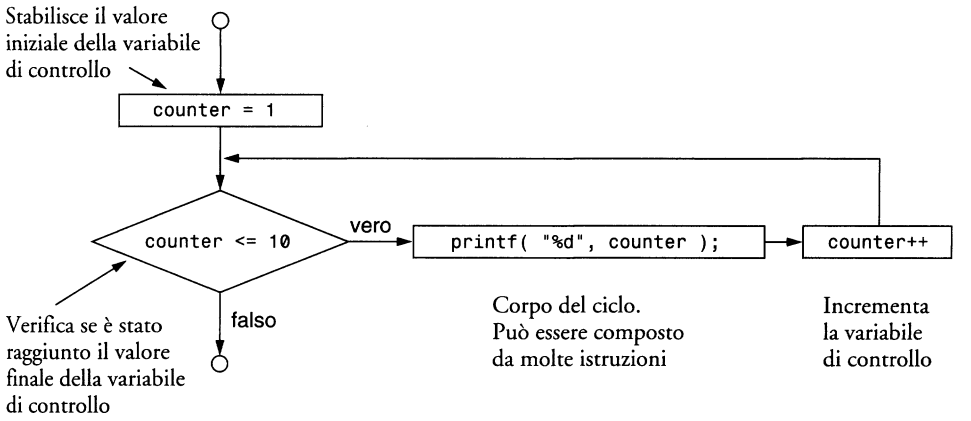


Figura 4.4 Il diagramma di flusso di un tipico comando di iterazione for

diagramma di flusso contiene soltanto dei rettangoli e un rombo. Immaginate, ancora una volta, che il programmatore possa accedere a un contenitore capiente di comandi `for` vuoti (rappresentati come segmenti di diagrammi di flusso). La quantità di questi è tale che, accatastandoli e nidificandoli con gli altri comandi di controllo, egli potrà formare una implementazione strutturata per il flusso di controllo di un algoritmo. I rettangoli e i rombi saranno quindi completati con le azioni e le decisioni appropriate per l'algoritmo.

Collaudo e messa a punto 4.3

Nonostante il valore della variabile di controllo possa essere modificato anche all'interno del corpo di un ciclo `for`*, ciò potrà generare degli errori logici difficili da individuare. Quindi sarà meglio non cambiare tale valore.*

4.6 Esempi di utilizzo del comando for

I seguenti esempi mostrano dei metodi per far variare la variabile di controllo in un comando `for`.

1. Far variare la variabile di controllo da 1 a 100 in incrementi di 1.

```
for ( i = 1; i <= 100; i++ )
```

2. Far variare la variabile di controllo da 100 a 1 in incrementi di -1 (decrementi di 1).

```
for ( i = 100; i >= 1; i-- )
```

3. Far variare la variabile di controllo da 7 a 77 a passi di 7.

```
for ( i = 7; i <= 77; i += 7 )
```

4. Far variare la variabile di controllo da 20 a 2 a passi di -2.

```
for ( i = 20; i >= 2; i -= 2 )
```

5. Far variare la variabile di controllo sulla seguente sequenza di valori: 2, 5, 8, 11, 14, 17, 20.

```
for ( j = 2; j <= 20; j += 3 )
```

6. Far variare la variabile di controllo sulla seguente sequenza di valori: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( j = 99; j >= 0; j -= 11 )
```

I prossimi due esempi forniranno alcune semplici applicazioni del comando `for`. Il programma della Figura 4.5 utilizza il comando `for` per sommare tutti gli interi pari da 2 a 100.

Osservate che il corpo del comando `for` nella Figura 4.5 in realtà potrebbe essere unito alla sezione più a destra della intestazione del `for`, utilizzando l'operatore virgola nel seguente modo:

```
for ( number = 2; number <= 100; sum += number, number += 2 )
   ; /* istruzione vuota */
```

L'inizializzazione `sum = 0` potrebbe anch'essa essere unita alla sezione di inizializzazione del `for`.

Buona abitudine 4.5

Le istruzioni che precedono un `for` *e quelle all'interno del suo corpo potranno spesso essere compattate nella intestazione del* `for`*, ma evitate di farlo poiché ciò renderà meno leggibile il programma.*

```
1  /* Fig. 4.5: fig04_05.c
2     Somma con for */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int sum = 0; /* inizializza la somma */
9     int number; /* numero che deve essere aggiunto alla somma */
10
11    for ( number = 2; number <= 100; number += 2 ) {
12       sum += number; /* aggiunge il numero alla somma */
13    } /* fine del comando for */
14
15    printf( "Sum is %d\n", sum ); /* visualizza la somma */
16
17    return 0; /* indica che il programma è terminato con successo */
18
19 } /* fine della funzione main */
```

```
Sum is 2550
```

Figura 4.5 Somma con for

Buona abitudine 4.6

Limitate a una sola riga, se possibile, la dimensione delle intestazioni di un comando di controllo.

Il prossimo esempio calcolerà l'interesse composto, utilizzando il comando `for`. Considerate la seguente enunciazione del problema:

Una persona investe $1000,00 in un libretto di risparmio, che rende il 5 per cento di interesse. Calcolate e visualizzate l'ammontare del denaro nel conto alla fine di ogni anno, per 10 anni, supponendo che tutto l'interesse sia stato lasciato in deposito sul conto. Utilizzate la seguente formula per determinare le suddette cifre:

$a = p(1 + r)n$

dove

p è l'ammontare dell'investimento iniziale (ovverosia il capitale)

r è il tasso di interesse annuale

n è il numero degli anni

a è l'ammontare nel deposito alla fine dell'anno n-esimo.

Questo problema richiederà un ciclo che eseguirà i calcoli indicati per ognuno dei 10 anni in cui il denaro rimarrà nel deposito. La soluzione è mostrata nella Figura 4.6. [*Nota*: usando molti compilatori C per UNIX, dovete includere l'opzione `-lm` (ad esempio, `cc -lm fig04_06.c`) al momento della compilazione del programma in Figura 4.6. Ciò ha l'effetto di collegare la libreria delle funzioni matematiche al programma.]

Il comando `for` eseguirà il corpo del ciclo 10 volte, variando una variabile di controllo da 1 a 10 in incrementi di 1. Il C non include un operatore per l'operazione di elevamento a potenza, tuttavia potremo utilizzare a questo scopo la funzione `pow` della libreria standard. La funzione `pow(x, y)` calcola il valore di `x` elevato alla `y`-esima potenza. Il `double` è un tipo di dato in virgola mobile molto simile al `float`, ma una variabile di tipo `double` può immagazzinare numeri molto più grandi, con una precisione più alta di un `float`. Osservate che, qualora venga utilizzata una funzione matematica come `pow`, dovrà essere incluso il file di intestazione `math.h` (riga 4). In realtà, questo programma funzionerebbe, ma male, anche senza l'inclusione di `math.h`. La funzione `pow` richiede due argomenti `double`. Osservate che `year` è però un intero. Il file `math.h` include appunto delle informazioni che indicano al compilatore di convertire il valore di `year` in una rappresentazione `double` temporanea, prima di richiamare la funzione. Tali informazioni sono contenute in una entità chiamata *prototipo di funzione* di `pow`. I prototipi di funzione saranno spiegati nel Capitolo 5. Nello stesso capitolo, forniremo un compendio di `pow` e delle altre funzioni matematiche della libreria.

```
1  /* Fig. 4.6: fig04_06.c
2     Calcolare l'interesse composto */
3  #include <stdio.h>
4  #include <math.h>
5
6  /* l'esecuzione del programma inizia dalla funzione main */
7  int main()
8  {
9     double amount; /* ammontare del deposito */
10    double principal = 1000.0; /* capitale iniziale */
11    double rate = 0.05; /* tasso di interesse annuo */
12    int year; /* contatore degli anni */
13
14    /* visualizza le intestazioni delle colonne della tabella */
15    printf( "%4s%21s\n", "Year", "Amount on deposit" );
16
17    /* calcola l'ammontare del deposito per ognuno dei dieci anni */
18    for ( year = 1; year <= 10; year++ ) {
19
20       /* calcola il nuovo ammontare per l'anno specificato */
21       amount = principal * pow( 1.0 + rate, year );
22
23       /* visualizza una riga della tabella */
24       printf( "%4d%21.2f\n", year, amount );
25    } /* fine del comando for */
26
27    return 0; /* indica che il programma è terminato con successo */
28
29 } /* fine della funzione main */
```

Figura 4.6 Calcolare l'interesse composto con for (continua)

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Figura 4.6 Calcolare l'interesse composto con for

Osservate che abbiamo dichiarato le variabili `amount`, `principal` e `rate` con il tipo `double`. Abbiamo fatto ciò per semplicità, perché avremo a che fare con parti frazionarie di dollari.

Collaudo e messa a punto 4.4

Non utilizzate variabili di tipo `float` *o* `double` *per eseguire calcoli finanziari. L'imprecisione dei numeri in virgola mobile potrà provocare degli errori che produrranno valori monetari non corretti. [Negli esercizi esploreremo l'utilizzo degli interi per eseguire i calcoli finanziari.]*

Ecco una semplice spiegazione di quello che potrebbe andar male, qualora si utilizzasse il tipo `float` o `double` per rappresentare degli importi in dollari.

Due quantitativi di dollari, espressi in `float` e immagazzinati nella macchina, potrebbero essere 14,234 (che sarebbe visualizzato come 14,23 con `%.2f`) e 18,673 (che con `%.2f` sarebbe stampato come 18,67). Nel momento in cui le suddette cifre saranno sommate, produrranno la somma 32,907 che, con `%.2f` sarebbe visualizzata come 32,91. Di conseguenza, la vostra visualizzazione potrebbe apparire come:

```
  14,23
+ 18,67
-------
  32,91
```

è però evidente che la somma dei singoli numeri, così come vengono visualizzati, debba essere 32,90! Siete stati avvisati!

La specifica di conversione `%21.2f` è stata utilizzata nel programma per visualizzare il valore della variabile `amount`. Il `21` nella specifica di conversione denota la *dimensione del campo* in cui il valore sarà visualizzato. Una dimensione di campo `21` specifica che il valore visualizzato apparirà in `21` posizioni di visualizzazione. Il `2` specifica la precisione (vale a dire, il numero di posizioni decimali). Qualora il numero dei caratteri visualizzati sia inferiore alla dimensione del campo, il valore sarà automaticamente *giustificato a destra* all'interno del campo. Ciò sarà particolarmente utile per allineare dei valori in virgola mobile con la stessa precisione (in modo che i loro separatori decimali si allineino verticalmente). Inserite un – (segno meno) tra il `%` e la dimensione del campo, per *giustificare a sinistra* il valore inserito in quel

campo. Osservate che il segno meno potrà anche essere utilizzato per giustificare a sinistra gli interi (come in `%-6d`), o le stringhe di caratteri (come in `%-8s`). Nel Capitolo 9 tratteremo in dettaglio le potenti capacità di formattazione di `printf` e `scanf`.

4.7 Il comando di selezione multipla switch

Nel Capitolo 3, abbiamo discusso del comando di selezione singola `if` e di quello di selezione doppia `if...else`. A volte però un algoritmo potrebbe richiedere che una data variabile od espressione venga confrontata distintamente con ognuno dei valori che essa può assumere e che, a seconda del risultato del confronto, vengano intraprese delle azioni distinte. Questo processo viene chiamato selezione multipla. Il C fornisce il comando di selezione multipla `switch` (interruttore) per gestire una tale serie di decisioni.

Il comando `switch` consiste di una serie di etichette `case` (caso) e di un caso opzionale `default`. Il programma nella Figura 4.7 utilizza `switch`, per contare il numero di ognuna delle lettere ottenute dagli studenti come votazione al termine di un esame.

```
1  /* Fig. 4.7: fig04_07.c
2     Contare le lettere dei voti */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int grade; /* un voto */
9     int aCount = 0; /* numero di A */
10    int bCount = 0; /* numero di B */
11    int cCount = 0; /* numero di C */
12    int dCount = 0; /* numero di D */
13    int fCount = 0; /* numero di F*/
14
15    printf( "Enter the letter grades.\n" );
16    printf( "Enter the EOF character to end input.\n" );
17
18    /* itera finché l'utente non digita la sequenza di tasti che
         codifica la "fine del file" */
19    while ( ( grade = getchar() ) != EOF ) {
20
21    /* determina che voto è stato inserito */
22    switch ( grade ) {    /* switch nidificato nel while */
23
24          case 'A': /* il voto era una A maiuscola */
25          case 'a':           /* o una a minuscola */
26             ++aCount; /* incrementa aCount */
27             break; /* necessario per uscire dallo switch */
28
29          case 'B': /* il voto era una B maiuscola */
30          case 'b':           /* o una b minuscola */
```

Figura 4.7 Un esempio di utilizzo di switch (continua)

```
            ++bCount; /* incrementa bCount */
            break; /* esce dallo switch */

         case 'C': /* il voto era una C maiuscola */
         case 'c':           /* o una c minuscola */
            ++cCount; /* incrementa cCount */
            break; /* esce dallo switch */

         case 'D': /* il voto era una D maiuscola */
         case 'd':           /* o una d minuscola */
            ++dCount; /* incrementa dCount */
            break; /* esce dallo switch */

         case 'F': /* il voto era una F maiuscola */
         case 'f':           /* o una f minuscola */
            ++fCount; /* incrementa fCount */
            break; /* esce dallo switch */

         case '\n': /* ignora i newline */
         case '\t': /* le tabulazioni */
         case ' ':  /* e gli spazi nell'input */
            break; /* esce dallo switch */

         default:        /* intercetta tutti gli altri caratteri */
            printf( "Incorrect letter grade entered." );
            printf( " Enter a new grade.\n" );
            break; /* opzionale: esce dallo switch in ogni caso */
      } /* fine del comando switch */

   } /* fine del comando while */

   /* visualizza il sommario dei risultati */
   printf( "\nTotals for each letter grade are:\n" );
   printf( "A: %d\n", aCount );
   printf( "B: %d\n", bCount );
   printf( "C: %d\n", cCount );
   printf( "D: %d\n", dCount );
   printf( "F: %d\n", fCount );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */
```

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
```

Figura 4.7 Un esempio di utilizzo di switch (continua)

```
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Figura 4.7 Un esempio di utilizzo di switch

Nel programma, l'utente immetterà le lettere di votazione che saranno state assegnate a una classe. All'interno della intestazione del `while` (riga 19),

```
while ( ( grade = getchar() ) != EOF )
```

l'assegnamento incluso nelle parentesi `( grade = getchar() )` sarà eseguito per primo. La funzione `getchar` (dalla libreria standard di input/output) leggerà un carattere dalla tastiera e lo immagazzinerà nella variabile intera `grade`. Normalmente i caratteri sono immagazzinati in variabili di tipo `char`. Una importante caratteristica del C è tuttavia che i caratteri possono essere immagazzinati in qualsiasi tipo di dato intero, poiché all'interno del computer sono rappresentati come interi di un byte. Di conseguenza, potremo trattare un carattere sia come intero sia come carattere vero e proprio, secondo il suo utilizzo. Per esempio, l'istruzione

```
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

utilizzerà le specifiche di conversione `%c` e `%d` per visualizzare rispettivamente il carattere `a` e il suo valore intero. Il risultato sarà:

```
The character (a) has the value 97.
```

L'intero 97 è la rappresentazione numerica del carattere nel computer. Molti computer utilizzano oggi l'*insieme di caratteri ASCII (American Standard Code for Information Interchange, Codice Americano Standard per lo Scambio delle Informazioni),* nel quale il 97 rappresenta la lettera minuscola `'a'`. Un elenco completo dei caratteri ASCII e dei loro valori decimali sarà presentato nell'Appendice D. I caratteri possono essere letti con `scanf`, utilizzando la specifica di conversione `%c`.

Le istruzioni di assegnamento, considerate nel loro complesso, hanno in realtà un valore. Questo è precisamente il valore che sarà assegnato alla variabile alla sinistra del segno `=`. Il valore dell'assegnamento `grade = getchar()` sarà il carattere restituito dalla `getchar` e assegnato alla variabile `grade`.

Il fatto che le istruzioni di assegnamento abbiano dei valori potrà essere utile per inizializzare diverse variabili con lo stesso valore. Per esempio,

```
a = b = c = 0;
```

valuterà in primo luogo l'assegnamento `c = 0` (poiché l'operatore = associa da destra a sinistra). Alla variabile `b` sarà quindi assegnato il valore dell'assegnamento `c = 0` (ovverosia 0). In seguito, alla variabile `a` sarà assegnato il valore dell'assegnamento `b = (c = 0)` (anch'esso 0). Nel programma, il valore dell'assegnamento `grade = getchar()` sarà confrontato con il valore di `EOF` (un simbolo il cui acronimo sta per "end of file", "fine del file"). Noi utilizzeremo `EOF` (che normalmente vale -1) come valore sentinella. L'utente premerà una combinazione di tasti dipendente dal sistema, per indicare la "fine del file", ovverosia "Non ho più dati da immettere". L'EOF è una costante simbolica intera definita nel file di intestazione `<stdio.h>` (vedremo come si definiscono le costanti simboliche nel Capitolo 6). Qualora il valore assegnato a `grade` sia uguale a `EOF`, il programma terminerà la propria esecuzione. In questo programma, abbiamo scelto di rappresentare i caratteri con degli `int`, poiché `EOF` ha un valore intero che, lo ripetiamo ancora una volta, normalmente è -1.

Obiettivo portabilità 4.1

Le combinazioni di tasti per immettere `EOF` *(fine del file) dipendono dal sistema!*

Obiettivo portabilità 4.2

Controllare la costante simbolica `EOF`*, invece che –1, renderà i programmi più portabili. Lo standard ANSI stabilisce che* `EOF` *debba essere un valore intero negativo, ma non necessariamente –1. Di conseguenza* `EOF` *potrebbe avere valori diversi su sistemi differenti.*

Sui sistemi UNIX e molti altri l'indicatore di **`EOF`** è immesso premendo la sequenza di tasti

<return> <ctrl-d>

Questa notazione indica di premere il tasto return (o enter, o invio) e quindi di premere simultaneamente entrambi i tasti `ctrl` e `d`. Su altri sistemi, come su Windows della Microsoft Corporation, l'indicatore di `EOF` può essere immesso premendo

<ctrl-z>

L'utente immetterà le votazioni alla tastiera. Nel momento in cui sarà stato premuto il tasto *Return* (o *Enter*, o invio), i caratteri saranno letti uno per volta dalla funzione `getchar`. Qualora il carattere immesso non sia uguale a `EOF`, si entrerà nel comando `switch` (riga 22). La parola chiave `switch` è seguita dal nome della variabile `grade` tra parentesi tonde. Questa è denominata *espressione di controllo*. Il valore di questa espressione sarà confrontato con ognuna delle *etichette case*. Supponete che l'utente abbia immesso come votazione la lettera `C`. Questa sarà automaticamente confrontata con ogni `case` dello `switch`. Nel caso sia stata verificata una corrispondenza (`case 'C':`), saranno eseguite le istruzioni di quel `case`. Nel caso della lettera `C`, `cCount` sarà incrementata di `1` (riga 36) e il comando `switch` sarà abbandonato immediatamente con l'istruzione `break`.

L'istruzione `break` induce il controllo del programma a continuare con la prima istruzione dopo il comando `switch`. L'istruzione `break` è utilizzata perché altrimenti sarebbero eseguiti insieme tutti i `case` di una istruzione `switch`. Nel caso in cui `break` non fosse utilizzata

in nessun posto del comando `switch`, allora ogniqualvolta fosse riscontrata una corrispondenza all'interno del comando, sarebbero eseguite anche le istruzioni di tutti i `case` rimanenti. Qualora non sia stata riscontrata nessuna occorrenza, sarà eseguito il caso `default` e sarà visualizzato un messaggio di errore.

Ogni `case` può contenere una o più azioni. Il comando `switch` è differente da tutti gli altri comandi, poiché in un `case` di `switch` non sono necessarie le parentesi graffe intorno alle azioni multiple. Il diagramma di flusso generico per un comando di selezione multipla `switch`, che utilizzi un `break` per ogni `case`, è mostrato nella Figura 4.8.

Il diagramma di flusso rende evidente che ogni istruzione `break`, alla fine di un `case`, fa in modo che il controllo esca immediatamente dal comando `switch`. Ancora una volta, osservate che, a parte i cerchietti e le frecce, il diagramma di flusso contiene solo rettangoli e rombi. Immaginate, ancora, che il programmatore abbia accesso a un contenitore capiente pieno di comandi `swicth` vuoti (rappresentati come segmenti di diagrammi di flusso). La quantità di questi è tale che il programmatore, accatastandoli e nidificandoli con gli altri comandi di controllo, possa formare una implementazione strutturata per il flusso di controllo di un algoritmo. I rettangoli e i rombi saranno quindi completati con le azioni e le decisioni appropriate all'algoritmo.

Errore tipico 4.5

Dimenticare una istruzione `break` *qualora ne sia necessaria una in un comando* `switch` *è un errore logico.*

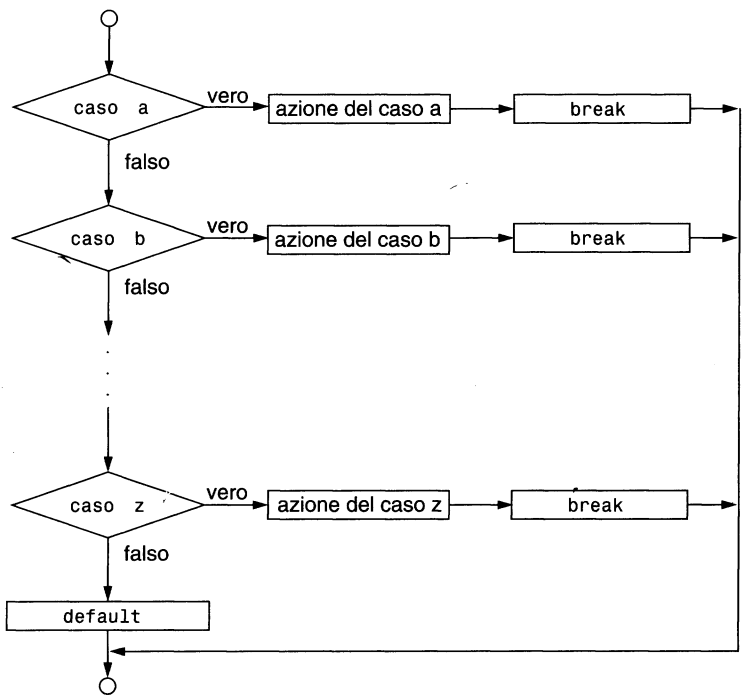


Figura 4.8 Il comando di selezione multipla switch con istruzioni break

Buona abitudine 4.7

Fornite un caso `default` *nelle istruzioni* `switch`. *Normalmente i casi non controllati in modo esplicito all'interno di uno* `switch` *saranno ignorati. Il caso* `default` *aiuterà a evitarlo, spingendo il programmatore a concentrarsi sulla necessità di elaborare le condizioni eccezionali. Ci sono situazioni in cui non sarà necessario nessuna elaborazione per il caso* `default`.

Buona abitudine 4.8

Sistemare il caso `default` *per ultimo è considerata una buona abitudine di programmazione, sebbene le clausole* `case` *e quella del caso* `default` *possano presentarsi in qualsiasi ordine all'interno di un comando* `switch`.

Buona abitudine 4.9

In un comando `switch`, *qualora la clausola* `default` *sia sistemata per ultima, l'istruzione* `break` *all'interno di quell'ultimo caso non sarà necessaria. Alcuni programmatori però includono ugualmente un* `break` *per motivi di chiarezza e simmetria con gli altri* `case`.

Nel comando `switch` della Figura 4.7, le righe

```
case '\n': /* ignora i newline */
case '\t': /* le tabulazioni */
case ' ':  /* e gli spazi nell'input */
   break; /* esce dallo switch */
```

indurranno il programma a ignorare i caratteri di newline e di spazio. Leggere i caratteri uno per volta può causare alcuni problemi. Per fare in modo che il programma legga i caratteri, questi dovranno essere inviati al computer premendo il *tasto invio*. Ciò provocherà l'inserimento nell'input del carattere newline, subito dopo quello che si desideri elaborare. Tale carattere di newline dovrà spesso essere trattato in modo speciale, per far funzionare correttamente il programma. Con l'inclusione dei casi precedenti, nel nostro comando `switch`, preverremo appunto la stampa del messaggio di errore nel caso `default`, ogni volta che nell'input sarà incontrato un newline o uno spazio.

Errore tipico 4.6

Non elaborare i caratteri di newline nell'input, quando si leggono dei caratteri uno per volta, potrà provocare degli errori logici.

Collaudo e messa a punto 4.5

Ricordate di fornire le istruzioni per l'elaborazione dei caratteri newline presenti nell'input, quando elaborate i caratteri uno per volta.

Osservate che le diverse etichette di caso sistemate insieme (come `case 'D': case 'd':` nella Figura 4.7) significano semplicemente che per entrambi i casi sarà eseguito lo stesso insieme di azioni.

Quando utilizzate il comando `switch`, ricordate che può essere utilizzato soltanto per controllare una *espressione intera costante*, ovverosia, ogni combinazione di costanti di carat-

tere e intere che possano essere valutate come valori interi. Una costante di tipo carattere è rappresentata con il carattere specifico posto tra apici singoli, come in `'A'`. Per essere riconosciuti come costanti di tipo carattere, questi devono essere racchiusi all'interno degli apici singoli. Le costanti intere sono semplicemente dei valori interi. Nel nostro esempio, abbiamo utilizzato delle costanti di tipo carattere. Ricordate che i caratteri sono in realtà dei valori interi memorizzati in un byte.

I linguaggi portabili, come il C, devono avere delle dimensioni flessibili per i tipi di dato. Applicazioni differenti possono avere bisogno di interi di dimensioni differenti. Il C fornisce diversi tipi di dato per rappresentare gli interi. L'intervallo dei valori interi per ognuno dei tipi forniti dipende dal particolare hardware del computer. In aggiunta ai tipi `int` e `char`, il C fornisce i tipi `short` (una abbreviazione per `short int`) e `long` (per `long int`). Il C specifica che il campo di variabilità minimo dei valori interi `short` deve essere ±32767. Per la gran maggioranza dei calcoli interi, saranno sufficienti degli interi `long`. Lo standard specifica che il campo di variabilità minimo dei valori `long` deve essere ±2147483647. Lo standard stabilisce che il campo di variabilità dei valori `int` deve essere grande almeno quanto quello degli interi `short`, ma che non può essere più grande di quello degli interi `long`. Il tipo di dato `char` potrà essere utilizzato per rappresentare degli interi nel campo di variabilità ±127, oppure ognuno dei caratteri facenti parte dell'insieme dei caratteri del computer.

4.8 Il comando di iterazione do...while

Il comando di iterazione `do...while` è simile al comando `while`. La condizione di continuazione del ciclo, nel comando `while`, è controllata all'inizio dello stesso, prima che sia eseguito il corpo dell'iterazione. Il comando `do...while` controlla la condizione di continuazione del ciclo *dopo* che è stato eseguito il corpo dello stesso. Di conseguenza, le istruzioni all'interno del corpo del ciclo saranno eseguite almeno una volta. Al termine di un'istruzione `do...while`, l'esecuzione continuerà con l'istruzione successiva alla clausola `while`. Osservate che nel comando `do...while` non sarà necessario utilizzare le parentesi graffe, qualora all'interno del corpo ci sia soltanto un'istruzione. Tuttavia, le parentesi graffe sono incluse di solito per evitare la confusione tra i comandi `while` e `do...while`. Per esempio,

```
while( condizione )
```

è normalmente vista come l'intestazione di un comando `while`. Un comando `do...while` senza parentesi graffe intorno a un corpo formato da una singola istruzione apparirebbe come

```
do
   istruzione
while( condizione );
```

che potrebbe creare confusione. L'ultima riga, `while( condizione );`, potrebbe essere fraintesa dal lettore come un comando `while` contenente un'istruzione vuota. Di conseguenza, per evitare confusioni un comando `do...while` con un'istruzione singola sarà spesso scritto nel modo seguente:

```
do {
   istruzione
} while ( condizione );
```

Buona abitudine 4.10

Alcuni programmatori includono sempre le parentesi graffe all'interno di un comando `do...while`, *anche quando non sono necessarie. Ciò aiuta a eliminare l'ambiguità tra il comando* `do...while` *contenente un'istruzione singola e il comando* `while`.

Errore tipico 4.7

I cicli infiniti sono provocati quando la condizione di continuazione del ciclo in un comando `while`, `for` *o* `do...while` *non diventa mai falsa. Per prevenirli, assicuratevi che non ci sia un punto e virgola immediatamente dopo l'intestazione di un comando* `while` *o* `for`. *In un ciclo controllato da un contatore, assicuratevi che la variabile di controllo sia incrementata (o decrementata) nel corpo del ciclo. In un ciclo controllato da un valore sentinella, assicuratevi che alla fine questo sia preso in input.*

Il programma nella Figura 4.9 utilizza un comando `do...while` per visualizzare i numeri da 1 a 10. Notate che la variabile di controllo `counter` è stata incrementata (con un preincremento) nella sezione per il controllo della continuazione del ciclo. Osservate anche l'utilizzo delle parentesi graffe per racchiudere il corpo del comando `do...while` formato da un'istruzione singola.

Il diagramma di flusso del comando `do...while` è mostrato nella Figura 4.10. Questo diagramma di flusso evidenzia che la condizione di continuazione del ciclo non sarà verificata, fintanto che l'azione non sarà stata eseguita almeno una volta. Ancora una volta, osservate che, a parte i cerchietti e le frecce, il diagramma di flusso contiene soltanto un rettangolo e un rombo. Immaginate, di nuovo, che il programmatore abbia accesso a un capiente contenitore di comandi `do...while` vuoti (rappresentati come segmenti di diagrammi di flusso). Il numero di questi è tale che il programmatore, accatastandoli e nidificandoli con gli altri comandi di controllo, possa formare un'implementazione strutturata per il flusso di controllo di un algoritmo. I rettangoli e i rombi saranno quindi completati con le azioni e le decisioni appropriate all'algoritmo.

```
1  /* Fig. 4.9: fig04_09.c
2     Usare il comando di iterazione do...while */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int counter = 1; /* inizializza il contatore */
9
10    do {
11       printf( "%d  ", counter ); /* visualizza il contatore */
12    } while (++counter <= 10); /* fine del comando do...while */
13
14    return 0; /* indica che il programma è terminato con successo */
15
16 } /* fine della funzione main */
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 4.9 Esempio di utilizzo del comando do...while

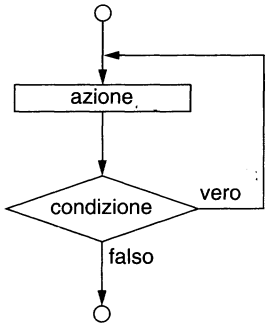


Figura 4.10 Diagramma di flusso del comando di iterazione do...while

4.9 Le istruzioni break e continue

Le istruzioni `break` e `continue` sono utilizzate per alterare il flusso di controllo. L'istruzione `break`, qualora sia eseguita in un comando `while`, `for`, `do...while` o `switch`, provocherà l'uscita immediata da quel comando. L'esecuzione del programma continuerà con la prima istruzione successiva al comando. L'utilizzo tipico dell'istruzione `break` è per anticipare l'uscita da un ciclo, oppure per ignorare la parte rimanente di un comando `switch` (come nella Figura 4.7). La Figura 4.11 dimostra l'utilizzo dell'istruzione `break` in un comando di iterazione `for`. L'istruzione `break` sarà eseguita quando il comando `if` avrà determinato che `x` avrà assunto il valore `5`. Ciò terminerà l'esecuzione della istruzione `for` e il programma potrà continuare con la funzione `printf` susseguente. Il ciclo sarà eseguito completamente soltanto quattro volte.

Qualora sia eseguita in un comando `while`, `for` o `do...while`, l'istruzione `continue` farà in modo che quelle rimanenti nel corpo del comando siano ignorate e che sia eseguita l'iterazione successiva del ciclo. Nei comandi `while` e `do...while`, la condizione di continuazione del ciclo sarà valutata immediatamente dopo l'esecuzione dell'istruzione `continue`. Nel comando `for`, prima sarà eseguita l'espressione di incremento e in seguito sarà valutata la condizione di continuazione del ciclo. In precedenza, abbiamo affermato che il comando `while` potrebbe essere utilizzato, nella maggior parte dei casi, per rappresentare il comando `for`. L'unica eccezione si verifica quando l'espressione di incremento all'interno del comando `while` segue l'istruzione `continue`. In questo caso, l'incremento non sarà eseguito prima della condizione di continuazione dell'iterazione e il `while` non sarà eseguito allo stesso modo del `for`. La Figura 4.12 utilizza l'istruzione `continue` in un comando `for`, per saltare l'istruzione `printf` nel comando e cominciare l'iterazione successiva del ciclo.

Ingegneria del software 4.2

Alcuni programmatori pensano che `break` *e* `continue` *violino le norme della programmazione strutturata. Dato che l'effetto di queste istruzioni potrà anche essere ottenuto da tecniche di programmazione strutturata, come impareremo presto, questi programmatori non utilizzano* `break` *e* `continue`.

```
1  /* Fig. 4.11: fig04_11.c
2     Usare l'istruzione break in un comando for */
3  #include <stdio.h>
4
```

Figura 4.11 Usare l'istruzione break in un comando for (continua)

```
/* l'esecuzione del programma inizia dalla funzione main */
int main()
{
   int x; /* contatore */

   /* itera 10 volte */
   for ( x = 1; x <= 10; x++ ) {

      /* se x vale 5, termina il ciclo */
      if ( x == 5 ) {
         break;    /* interrompe il ciclo solo se x vale 5 */
      } /* fine del comando if */

      printf( "%d ", x ); /* visualizza il valore di x */
   } /* fine del comando for */

   printf( "\nBroke out of loop at x == %d\n", x );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */
```

```
1 2 3 4
Broke out of loop at x == 5
```

Figura 4.11 Usare l'istruzione break in un comando for

```
/* Fig. 4.12: fig04_12.c
   Usare l'istruzione continue in un comando for */
#include <stdio.h>

/* l'esecuzione del programma inizia dalla funzione main */
int main()
{
   int x; /* contatore */

   /* itera 10 volte */
   for ( x = 1; x <= 10; x++ ) {

      /* se x vale 5, continua con la prossima iterazione del ciclo */
      if ( x == 5 ) {
         continue;  /* salta il codice restante del ciclo */
      }  /* fine del comando if */

      printf( "%d ", x ); /* visualizza il valore di x */
   } /* fine del comando for */

   printf( "\nUsed continue to skip printing the value 5\n" );
```

Figura 4.12 Usare l'istruzione continue in un comando for (continua)

```
22
23      return 0; /* indica che il programma è terminato con successo */
24
25   } /* fine della funzione main */
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Figura 4.12 Usare l'istruzione continue in un comando for

Obiettivo efficienza 4.1

Le istruzioni `break` *e* `continue`*, quando sono utilizzate in modo appropriato, saranno eseguite più velocemente delle corrispondenti tecniche strutturate che apprenderemo presto.*

Ingegneria del software 4.3

C'è una dicotomia tra la ricerca di una progettazione di software di qualità e quella di software più efficiente. Spesso uno di questi obiettivi è raggiunto a discapito dell'altro.

4.10 Gli operatori logici

Fino a questo punto abbiamo studiato soltanto delle *condizioni semplici* come `counter <= 10`, `total > 1000` e `number != sentinelValue`. Abbiamo espresso queste condizioni con l'uso degli operatori relazionali >, <, >=, <= e di quelli di uguaglianza == e !=. Ogni decisione ha verificato precisamente una condizione. Se avessimo voluto controllare delle condizioni multiple in una decisione, avremmo dovuto eseguire i suddetti controlli con delle istruzioni distinte o con dei comandi `if` o `if...else` nidificati.

Il C fornisce degli *operatori logici* che possono essere utilizzati per formare condizioni più complesse, combinando quelle semplici. Gli operatori logici sono: `&&` *(AND logico)*, `||` *(OR logico)* e `!` *(NOT logico* detto anche *negazione logica)*. Considereremo degli esempi per ognuno dei suddetti.

Supponete che, prima di scegliere un determinato percorso di esecuzione, vogliamo assicurarci che in un certo punto del programma due condizioni siano *entrambe* vere. In questo caso, potremmo utilizzare l'operatore logico `&&` come segue:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

Questa istruzione `if` contiene due condizioni semplici. La condizione `gender == 1` potrà essere valutata, per esempio, per determinare se una persona sia di sesso femminile. La condizione `age >= 65` sarà valutata per determinare se una certa persona sia un cittadino anziano. Le due condizioni semplici saranno valutate per prime perché le priorità di == e >= sono più alte di quella di `&&`. In seguito, l'istruzione `if` considererà la condizione combinata

```
gender == 1 && age >= 65
```

Questa condizione sarà vera se e solo se entrambe le condizioni semplici saranno vere. Infine, se la suddetta condizione combinata sarà effettivamente vera, allora il contatore `seniorFemales` sarà incrementato di `1`. Qualora una o entrambe le condizioni semplici siano false, il programma ignorerà l'incremento e procederà con l'istruzione successiva al comando `if`.

La Figura 4.13 descrive l'operatore &&. La tabella mostra tutte le quattro possibili combinazioni di valori zero (falso) e diversi da zero (vero) per espressione1 ed espressione2. Le tabelle di questo genere sono spesso chiamate *tabelle di verità*. Il C valuta 0 o 1 tutte le espressioni che includano degli operatori relazionali, di uguaglianza e/o logici. Sebbene il C imposti il valore vero con 1, esso accetta come vero *ogni* valore diverso da zero.

espressione1	espressione2	espressione1 && espressione2
0	0	0
0	diverso da zero	0
diverso da zero	0	0
diverso da zero	diverso da zero	1

Figura 4.13 Tabella di verità per l'operatore && (AND logico)

Consideriamo ora l'operatore || (OR logico). Supponete che, prima di scegliere un certo percorso di esecuzione, vogliamo assicurarci che a un certo punto del programma una *o* entrambe le condizioni siano vere. In questo caso useremo l'operatore ||, come nel seguente frammento di programma:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   printf( "Student grade is A\n" );
```

Questa istruzione contiene anch'essa due condizioni semplici. La condizione `semesterAverage >= 90` sarà valutata per determinare se lo studente del corso meriti una "A", grazie a un rendimento costante nel corso del semestre. La condizione `finalExam >= 90` sarà valutata per determinare se lo studente del corso meriti una "A", a causa di un rendimento straordinario nell'esame finale. In seguito, l'istruzione **if** considererà la condizione combinata

```
semesterAverage >= 90 || finalExam >= 90
```

e premierà lo studente con una "A" qualora una o entrambe le condizioni semplici siano vere. Osservate che il messaggio "`Student grade is A`" non sarà visualizzato solo quando entrambe le condizioni semplici saranno false (zero). La Figura 4.14 è una tabella di verità per l'operatore logico OR (||).

espressione1	espressione2	espressione1 || espressione2
0	0	0
0	diverso da zero	1
diverso da zero	0	1
diverso da zero	diverso da zero	1

Figura 4.14 Tabella di verità per l'operatore || (OR logico)

L'operatore && ha una priorità maggiore di ||. Entrambi gli operatori associano da sinistra a destra. Una espressione contenente gli operatori && o || sarà valutata soltanto fintanto che non sarà nota la sua verità o falsità. Di conseguenza, la valutazione della condizione

```
gender == 1 && age >= 65
```

si fermerà, qualora `gender` non sia uguale a `1` (a quel punto, l'intera espressione sarà sicuramente falsa), mentre continuerà, qualora `gender` sia uguale a `1` (poiché l'intera espressione potrebbe ancora essere vera se `age >= 65`).

Obiettivo efficienza 4.2

Nelle espressioni che utilizzano l'operatore `&&`*, fate in modo che la prima condizione sia quella che avrà più probabilità di essere falsa. Nelle espressioni che utilizzano l'operatore* `||`*, fate in modo che la prima condizione sia quella che avrà più probabilità di essere vera. Ciò potrà ridurre il tempo di esecuzione del programma.*

Il C fornisce `!` (la negazione logica) per consentire al programmatore di "invertire" il valore di una condizione. A differenza degli operatori `&&` e `||`, che combinano due condizioni (e sono quindi degli operatori binari), quello di negazione logica ha solo una singola condizione come operando (ed è quindi un operatore unario). L'operatore di negazione logica andrà inserito prima di una condizione, nel caso in cui siamo interessati a scegliere un percorso di esecuzione quando la condizione originale (senza l'operatore di negazione logica) sia falsa, come nel seguente frammento di codice:

```
if ( !( grade == sentinelValue ) )
   printf( "The next grade is %f\n", grade );
```

Le parentesi intorno alla condizione `grade == sentinelValue` sono necessarie perché l'operatore di negazione logica ha una priorità maggiore dell'operatore di uguaglianza. La Figura 4.15 è una tabella di verità per l'operatore di negazione logica.

Nella maggior parte dei casi, il programmatore potrà evitare di utilizzare la negazione logica, esprimendo la condizione in modo diverso, con un appropriato operatore relazionale. Per esempio, l'istruzione precedente potrà essere scritta anche nel modo seguente:

```
if (grade != sentinelValue)
   printf("The next grade is %f\n", grade);
```

La tabella nella Figura 4.16 mostra la priorità e l'associatività dei vari operatori introdotti sino a questo punto. Gli operatori sono mostrati dall'alto in basso in ordine decrescente di priorità.

espressione	!espressione
0	1
diverso da zero	0

Figura 4.15 Tabella di verità per l'operatore ! (negazione logica)

Operatori	Associatività	Tipo
`++ -- + - ! (tipo)`	da destra a sinistra	unari
`* / %`	da sinistra a destra	moltiplicativi
`+ -`	da sinistra a destra	additivi
`< <= > >=`	da sinistra a destra	relazionali

Figura 4.16 Priorità e associatività degli operatori (continua)

Operatori	Associatività	Tipo
== !=	da sinistra a destra	di uguaglianza
&&	da sinistra a destra	AND logico
||	da sinistra a destra	OR logico
?:	da destra a sinistra	condizionale
= += -= *= /= %=	da destra a sinistra	di assegnamento
,	da sinistra a destra	virgola

Figura 4.16 Priorità e associatività degli operatori

4.11 Confondere gli operatori di uguaglianza (==) e di assegnamento (=)

C'è un tipo di errore che i programmatori C, indipendentemente da quanto siano esperti, tendono a commettere così frequentemente che l'abbiamo ritenuto meritevole di una sezione distinta. Tale errore consiste nello scambiare accidentalmente gli operatori == (uguaglianza) e = (assegnamento). Ciò che rende questi scambi così dannosi è che essi, solitamente, non provocano degli errori di sintassi. Di solito, infatti, le istruzioni che contengono questi errori sono compilate correttamente e il programma sarà eseguito fino al suo completamento, generando probabilmente dei risultati non corretti a causa di errori logici, durante la fase di esecuzione.

Ci sono due aspetti del C che causano questi problemi. Uno è che, nel linguaggio C, ogni espressione che produca un valore può essere utilizzata nella sezione di decisione di ogni comando di controllo. L'espressione sarà trattata come falsa qualora il suo valore sia 0, mentre sarà considerata vera nel caso che il valore sia diverso da zero. Il secondo aspetto è che nel linguaggio C gli assegnamenti producono un valore, ovverosia quello che sarà assegnato alla variabile a sinistra dell'operatore di assegnamento. Per esempio, supponete che vogliamo scrivere:

```
if ( payCode == 4 )
   printf( "You get a bonus!" );
```

ma che accidentalmente scriviamo

```
if ( payCode = 4 )
   printf( "You get a bonus!" );
```

La prima istruzione **if** assegnerebbe correttamente un bonus alla persona il cui codice di pagamento è uguale a 4. La seconda istruzione **if** invece, quella con l'errore, valuterebbe l'espressione di assegnamento all'interno della sua condizione. Questa espressione è un assegnamento semplice il cui valore è la costante 4. Dato che ogni valore diverso da zero è interpretato come "vero", la condizione di questa istruzione **if** sarebbe considerata vera e la persona riceverebbe sempre un bonus, indipendentemente da quale sia il suo reale codice di pagamento!

Errore tipico 4.8

Usare l'operatore == in un assegnamento o utilizzare l'operatore = in una condizione di uguaglianza è un errore logico.

Normalmente i programmatori scrivono le condizioni in questo modo `x == 7`, con il nome della variabile a sinistra e la costante a destra. Invertendo le loro posizioni, facendo cioè in modo che la costante sia a sinistra e il nome della variabile a destra, come in `7 == x`, il programmatore che sostituisse accidentalmente l'operatore `==` con `=` sarebbe protetto dal compilatore. Il compilatore, infatti, lo tratterebbe come un errore di sintassi, poiché sul lato sinistro di una istruzione di assegnamento può essere sistemato soltanto il nome di una variabile. Questo eviterà almeno la potenziale devastazione di un errore logico durante l'esecuzione.

I nomi delle variabili sono detti anche *lvalue* (per "left value", valore di sinistra) proprio perché possono essere utilizzati sul lato sinistro di un operatore di assegnamento. Le costanti sono dette invece *rvalue* (per "right value", valore di destra) proprio perché possono essere utilizzate soltanto sul lato destro di un operatore di assegnamento. Osservate che un lvalue può anche essere utilizzato come rvalue, ma non viceversa.

Buona abitudine 4.11

Quando una espressione di uguaglianza contiene una variabile e una costante, come in `x == 1`, *alcuni programmatori preferiscono scriverla con la costante a sinistra e la variabile a destra, per proteggersi dall'errore logico che si verificherebbe, qualora sostituissero accidentalmente l'operatore* `==` *con* `=`.

L'altro lato della medaglia potrebbe essere altrettanto spiacevole. Supponete che il programmatore voglia assegnare un valore a una variabile con una semplice istruzione come:

```
x = 1;
```

ma scriva invece

```
x == 1;
```

Anche in questo caso, non si tratta di un errore di sintassi. Il compilatore valuterà piuttosto candidamente l'espressione condizionale. Qualora `x` fosse uguale a `1`, la condizione sarebbe considerata vera e l'espressione restituirebbe il valore `1`. Qualora `x` non fosse uguale a `1`, la condizione sarebbe considerata falsa e l'espressione restituirebbe il valore `0`. Indipendentemente da quale valore sarà restituito, non ci sarà nessun operatore di assegnamento, perciò il valore andrà semplicemente perso, e quello di `x` resterà inalterato, causando probabilmente un errore logico durante l'esecuzione. Sfortunatamente non abbiamo a disposizione un trucco pratico per aiutarvi con questo problema!

Collaudo e messa a punto 4.6

Dopo aver scritto un programma, eseguite una ricerca testuale di ogni simbolo `=` *e controllate che sia utilizzato correttamente.*

4.12 Riassunto della programmazione strutturata

I programmatori dovrebbero progettare i programmi proprio come gli architetti progettano i palazzi, impiegando la sapienza collettiva della loro categoria. Il nostro campo è però più giovane di quanto lo sia l'architettura e la nostra sapienza collettiva è considerevolmente più limitata. Abbiamo appreso una gran mole di cose in solo cinquant'anni. Forse, cosa ancora più importante, abbiamo appreso che la programmazione strutturata produce programmi che sono più semplici (rispetto a quelli non strutturati) da comprendere e quindi da collauda-

re, mettere a punto, modificare e anche da dimostrare corretti nel senso matematico del termine.

I Capitoli 3 e 4 si sono soffermati sui comandi di controllo del C. Ogni comando è stato presentato, rappresentato in diagrammi di flusso e discusso separatamente con degli esempi. Ora ricapitoleremo i risultati dei Capitoli 3 e 4 e introdurremo un semplice insieme di regole per la costruzione di programmi strutturati e per le loro proprietà.

La Figura 4.17 riassume i comandi di controllo discussi nei Capitoli 3 e 4. I cerchietti sono utilizzati nella figura per indicare gli unici punti di entrata e di uscita di ogni comando. Collegare arbitrariamente i singoli simboli di un diagramma di flusso potrà condurre a pro-

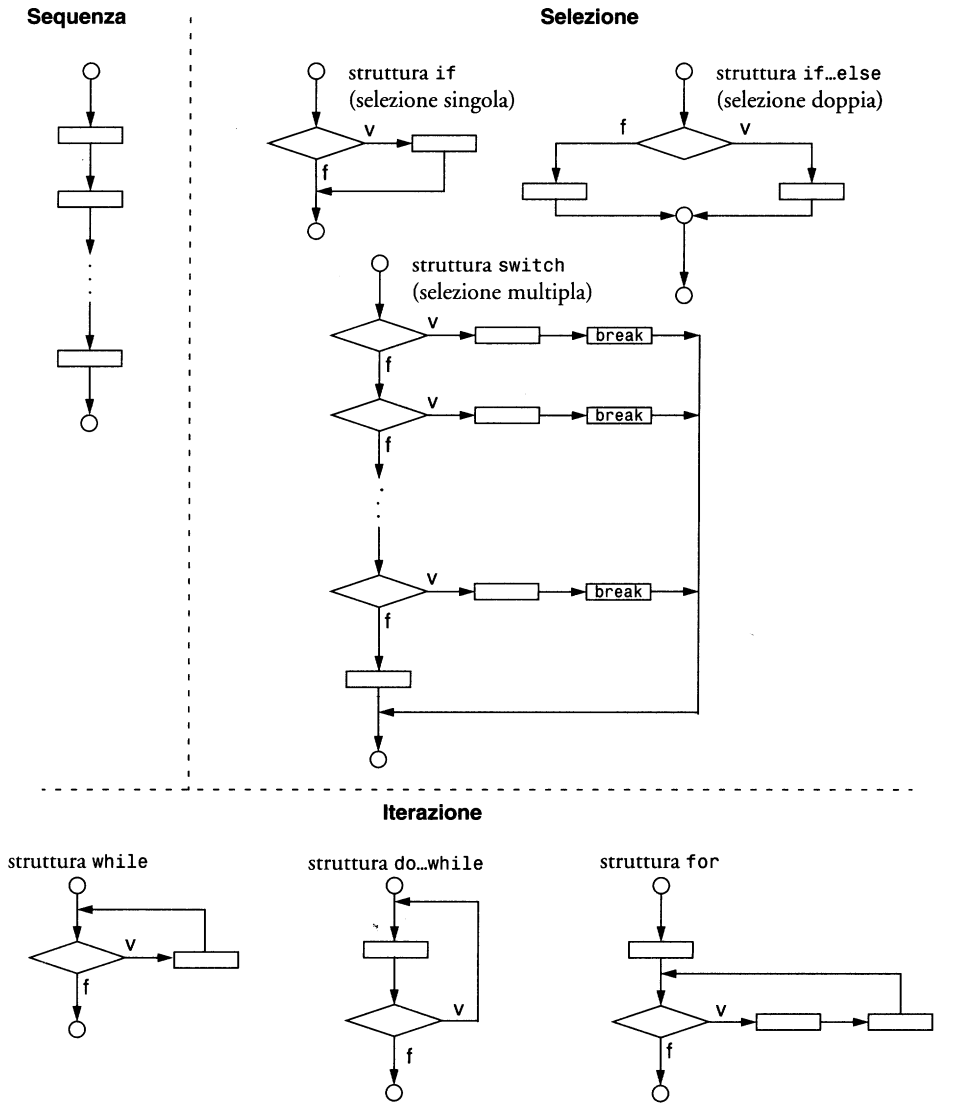


Figura 4.17 I comandi di sequenza, di selezione e di iterazione del C con un solo ingresso e una sola uscita

grammi non strutturati. Di conseguenza, la categoria dei programmatori ha scelto di combinare i simboli per i diagrammi di flusso, in modo tale da formare un insieme limitato di comandi di controllo e di costruire soltanto programmi strutturati, combinando appropriatamente i comandi di controllo secondo due sole semplici modalità. Per semplicità, sono stati utilizzati soltanto comandi di controllo con una singola entrata e uscita: c'è solo un modo per entrare e una sola maniera per uscire dal comando di controllo. Collegare i comandi di controllo in sequenza, in modo da formare dei programmi strutturati, è semplice: il punto di uscita di un comando di controllo è connesso direttamente al punto di entrata del comando di controllo successivo. In altre parole, i comandi di controllo saranno sistemati in un programma unicamente l'uno dopo l'altro: abbiamo perciò chiamato questo processo "accatastamento dei comandi di controllo". Le regole di formazione dei programmi strutturati consentono anche la nidificazione dei comandi di controllo.

La Figura 4.18 mostra le regole per formare appropriatamente dei programmi strutturati. Le regole assumono che ogni simbolo di rettangolo nei diagrammi di flusso possa essere utilizzato per indicare qualsiasi azione, incluse quelle di input/output.

L'applicazione delle regole della Figura 4.18 produrrà un diagramma di flusso strutturato con un aspetto ordinato e simile ai mattoncini delle costruzioni. Applicando ripetutamente la regola 2 al diagramma di flusso elementare (Figura 4.19), se ne produrrà uno strutturato contenente molti rettangoli in sequenza (Figura 4.20). Osservate che la regola 2 genererà una catasta di comandi di controllo; per questo motivo la chiameremo *regola di accatastamento.*

La regola 3 è detta *regola di nidificazione.* Applicando ripetutamente la regola 3 al diagramma di flusso elementare, se ne produrrà uno con comandi di controllo nidificati in modo ordinato. Per esempio, nella Figura 4.21, il rettangolo del diagramma di flusso elementare sarà prima sostituito da un comando di selezione doppia (`if...else`). In seguito la regola 3 sarà applicata di nuovo a entrambi i rettangoli del comando di selezione doppia, sostituendo ognuno di quei rettangoli con altri comandi di selezione doppia. I rettangoli tratteggiati intorno a ognuno dei comandi di selezione doppia rappresentano i rettangoli che sono stati sostituiti nel diagramma di flusso originale.

Regole per la formazione di programmi strutturati

1) Incominciare con il "diagramma di flusso elementare" (Figura 4.19).
2) Ogni rettangolo (azione) può essere sostituito da due rettangoli (azioni) in sequenza.
3) Ogni rettangolo (azione) può essere sostituito da un qualsiasi comando di controllo (sequenza, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4) Le regole 2 e 3 possono essere applicate ripetutamente e in qualsiasi ordine.

Figura 4.18 Le regole per la formazione di programmi strutturati

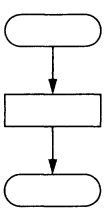


Figura 4.19 Il diagramma di flusso elementare

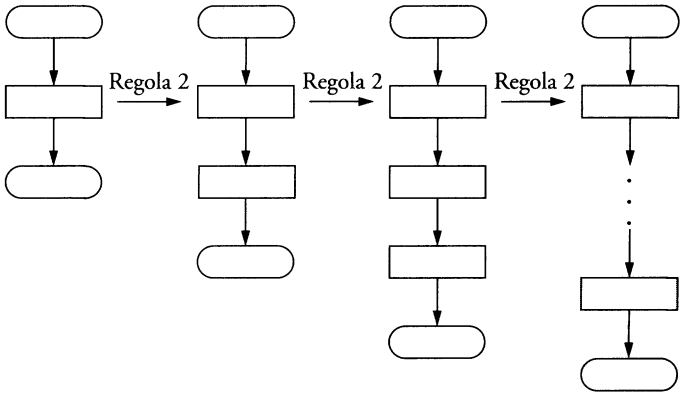


Figura 4.20 Applicare ripetutamente la regola 2 della Figura 4.18 al diagramma di flusso elementare

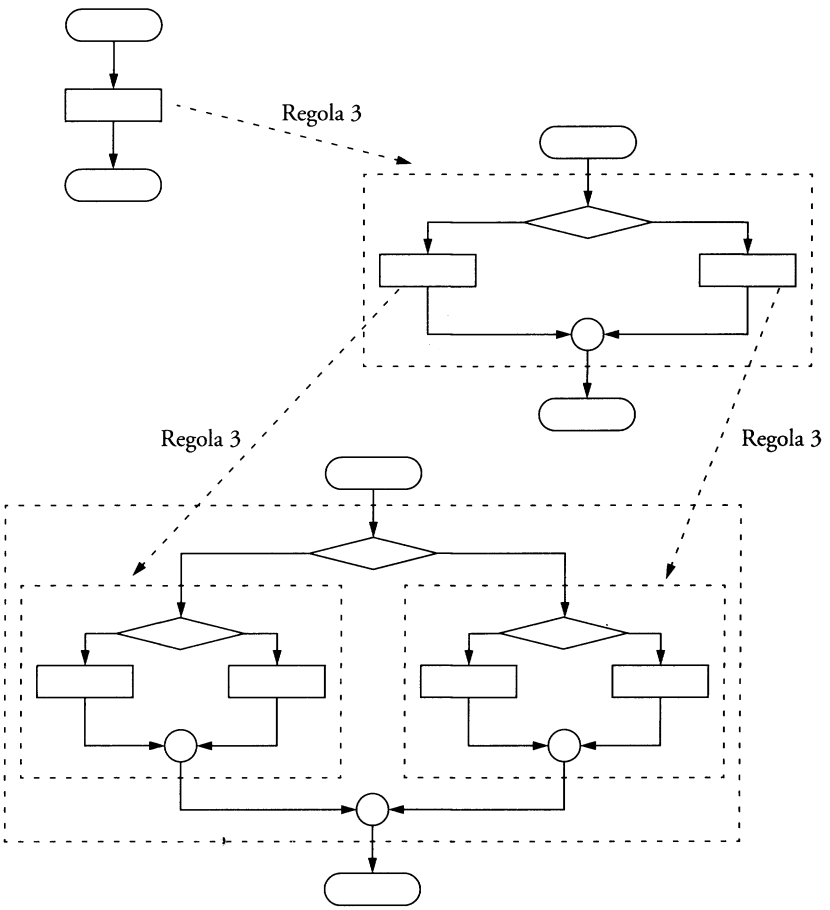


Figura 4.21 Applicazione della regola 3 della Figura 4.18 al diagramma di flusso elementare

La regola 4 genererà delle strutture più grandi, più complesse e più profondamente nidificate. I diagrammi di flusso che possono essere formati, applicando le regole della Figura 4.18, costituiscono l'insieme di tutti i possibili diagrammi di flusso strutturati e, quindi, l'insieme di tutti i possibili programmi strutturati.

È proprio grazie all'eliminazione della istruzione `goto` che questi mattoncini di costruzione non si sovrappongono mai l'uno sull'altro. La bellezza dell'approccio strutturato è che utilizziamo soltanto un piccolo numero di semplici pezzi, con un singolo ingresso e una sola uscita, e li assembliamo soltanto in due semplici maniere. La Figura 4.22 mostra i tipi di costruzioni, ottenute con l'accatastamento dei mattoncini, che si possono formare attraverso l'applicazione della regola 2 e i tipi di costruzioni, ottenute con l'accatastamento dei mattoncini, che non possono apparire nei diagrammi di flusso strutturati (grazie alla eliminazione della istruzione `goto`).

Se le regole della Figura 4.18 sono rispettate, un diagramma di flusso non strutturato (come quello della Figura 4.23) non potrà essere creato. Se non siete sicuri che un particolare diagramma di flusso sia strutturato, applicate le regole della Figura 4.18 all'inverso in modo da provare a ridurre il diagramma di flusso a quello elementare. Se il diagramma di flusso può essere ridotto a quello elementare, allora il diagramma di flusso originale è strutturato; altrimenti, non lo è.

La programmazione strutturata promuove la semplicità. Böhm e Jacopini hanno dimostrato che sono necessarie solo tre forme di controllo:

- Sequenza
- Selezione
- Iterazione

La sequenza è banale. La selezione è implementata in uno dei tre modi:

- comando `if` (selezione semplice)
- comando `if...else` (selezione doppia)
- comando `switch` (selezione multipla)

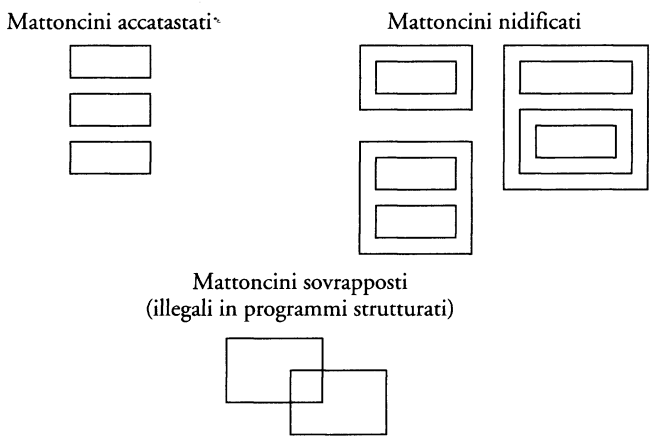


Figura 4.22 Tipi di costruzioni accatastate, nidificate e sovrapposte

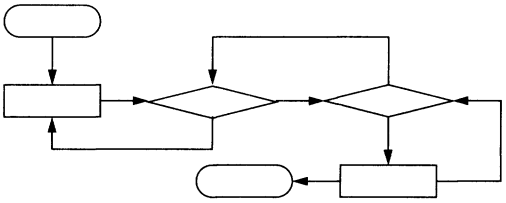


Figura 4.23 Un diagramma di flusso non strutturato

Di fatto, è elementare dimostrare che il semplice comando `if` è sufficiente a fornire ogni forma di selezione: tutto ciò che possa essere fatto con i comandi `if...else` e `switch` potrà essere implementato con il comando `if`.

L'iterazione è implementata in uno dei tre modi:

- comando `while`
- comando `do...while`
- comando `for`

È elementare provare che il comando `while` è sufficiente a fornire ogni forma di iterazione. Tutto ciò che possa essere fatto con i comandi `do...while` e `for` potrà essere implementato con il comando `while`.

La combinazione di questi risultati dimostra che qualsiasi forma di controllo di cui si potrà aver bisogno in un programma C potrà essere espressa con tre sole di esse:

- sequenza
- comando `if` (selezione)
- comando `while` (iterazione)

Questi comandi di controllo potranno essere combinati in due soli modi: accatastandoli e nidificandoli. La programmazione strutturata promuove di certo la semplicità.

Nei Capitoli 3 e 4, abbiamo discusso di come comporre i programmi con comandi di controllo contenenti azioni e decisioni. Nel Capitolo 5, introdurremo un'altra unità di strutturazione del programma chiamata *funzione*. Apprenderemo come costruire dei programmi complessi combinando le funzioni, che saranno formate a loro volta da comandi di controllo. Discuteremo anche del modo in cui l'utilizzo delle funzioni promuove la riusabilità del software.

Esercizi di autovalutazione

4.1 Riempite gli spazi in ognuna delle seguenti frasi.

a) Una iterazione controllata da un contatore è nota anche come iterazione _____, perché è noto in anticipo il numero di volte che il ciclo sarà eseguito.

b) Una iterazione controllata da un valore sentinella è nota anche come iterazione _____, perché non è noto in anticipo il numero di volte che il ciclo sarà ripetuto.

c) Nelle ripetizioni controllate da un contatore si utilizza una _____ per contare il numero di volte che un gruppo di istruzioni dovrà essere ripetuto.

d) L'istruzione _____, quando eseguita in un comando di iterazione, indurrà l'esecuzione immediata della iterazione successiva del ciclo.
e) L'istruzione _____, quando eseguita in un comando di iterazione o in uno `switch`, causerà l'uscita immediata dal comando.
f) La _____ è utilizzata per controllare una particolare variabile o espressione per ognuno dei valori costanti interi che essa potrà assumere.

4.2 Stabilite se le seguenti affermazioni sono vere o false. Qualora la risposta sia falsa, spiegatene il motivo.

a) Il caso `default` è obbligatorio nel comando di selezione `switch`.
b) L'istruzione `break` è obbligatoria nel caso `default` di un comando di selezione `switch`.
c) L'espressione `(x > y && a < b)` sarà considerata vera qualora lo sia `x > y` oppure qualora lo sia `a < b`.
d) Un'espressione contenente l'operatore `||` sarà considerata vera se uno o entrambi i suoi operandi sono veri.

4.3 Scrivete una istruzione o un loro insieme per eseguire ognuno dei seguenti compiti:

a) Sommate gli interi dispari tra 1 e 99 utilizzando un comando `for`. Supponete che siano state dichiarate le variabili `sum` e `count`.
b) Visualizzate il valore `333,546372` in un campo lungo `15` caratteri con le precisioni `1`, `2`, `3`, `4` e `5`. Giustificate a sinistra l'output. Quali saranno i cinque valori visualizzati?
c) Calcolate il valore di `2,5` elevato alla potenza `3` utilizzando la funzione `pow`. Visualizzate il risultato con precisione `2` in un campo con dimensione pari a `10` posizioni. Quale sarà il valore visualizzato?
d) Visualizzate gli interi da 1 a 20 utilizzando un ciclo `while` e la variabile contatore `x`. Supponete che `x` sia già stata dichiarata, ma non inizializzata. Visualizzate solo 5 interi per riga. [*Suggerimento*: usate il calcolo `x % 5`. Stampate un carattere newline quando il valore di questo calcolo sarà 0, altrimenti visualizzate un carattere di tabulazione.
e) Ripetete l'Esercizio 4.3 (d), utilizzando un comando `for`.]

4.4 Trovate l'errore in ognuno dei seguenti frammenti di codice e spiegate come correggerlo.

a)
```
x = 1;
while ( x <= 10 );
   x++;
}
```
b)
```
for ( y = .1; y != 1.0; y += .1 )
   printf( "%f\n", y );
```
c)
```
switch ( n ) {
case 1:
   printf( "The number is 1\n" );
case 2:
   printf( "The number is 2\n" );
   break;
default:
   printf( "The number is not 1 or 2\n" );
   break;
}
```
d) Il codice seguente dovrebbe visualizzare i valori da 1 a 10.
```
n = 1;
while ( n < 10 )
   printf( "%d ", n++ );
```

Risposte agli esercizi di autovalutazione

4.1 a) definita. b) indefinita. c) variabile di controllo o contatore. d) **`continue`**. e) `break`. f) comando di selezione `switch`.

4.2 a) Falso. Il caso `default` è opzionale. Se non è necessaria nessuna azione di `default`, allora non ci sarà nessun bisogno di un caso `default`.

b) Falso. L'istruzione `break` è utilizzata per uscire dal comando `switch`. L'istruzione `break` non sarà richiesta qualora il caso `default` sia l'ultimo.

c) Falso. Quando si utilizza l'operatore `&&`, entrambe le espressioni relazionali dovranno essere vere perché l'intera espressione sia considerata tale.

d) Vero.

4.3 a)
```
sum = 0;
for ( count = 1; count <= 99; count += 2 )
   sum += count;
```

b)
```
printf( "%-15.1f\n", 333.546372 ); /* visualizza 333,5      */
printf( "%-15.2f\n", 333.546372 );  /* visualizza 333,55     */
printf( "%-15.3f\n", 333.546372 );  /* visualizza 333,546    */
printf( "%-15.4f\n", 333.546372 );  /* visualizza 333,5464   */
printf( "%-15.5f\n", 333.546372 );  /* visualizza 333,54637  */
```

c)
```
printf( "%10.2f\n", pow( 2.5, 3 ) );     /* visualizza 15,63      */
```

d)
```
x = 1;
while ( x <= 20 ) {
   printf( "%d", x );
   if ( x % 5 == 0 )
      printf( "\n" );
   else
      printf( "\t" );
   x++;
}
```

o

```
x = 1;
while ( x <= 20 )
   if ( x % 5 == 0 )
      printf( "%d\n", x++ );
   else
      printf( "%d\t", x++ );
```

o

```
x = 0;
while ( ++x <= 20 ) {
   if ( x % 5 == 0 )
      printf( "%d\n", x );
   else
      printf( "%d\t", x );
```

e)
```
for ( x = 1; x <= 20; x++ ) {
printf( "%d", x );
   if ( x % 5 == 0 )
      printf( "\n" );
   else
      printf( "\t" );
}
```

o

```
for ( x = 1; x <= 20; x++ )
   if ( x % 5 == 0 )
      printf( "%d\n", x );
   else
      printf( "%d\t", x );
```

4.4 a) Errore: il punto e virgola dopo l'intestazione del `while` provocherà un ciclo infinito.
Correzione: sostituite il punto e virgola con una { oppure rimuovete il ; e la }.

b) Errore: l'utilizzo di un numero in virgola mobile per controllare il comando di iterazione `for`.
Correzione: utilizzate un intero ed eseguite i calcoli appropriati per ottenere il valore che desiderate.

```
for ( y = 1; y != 10; y++ )
   printf( "%f\n", ( float ) y / 10 );
```

c) Errore: manca l'istruzione `break` tra quelle del primo `case`.
Correzione: aggiungete un `break` alla fine delle istruzioni del primo `case`. Osservate che questo potrebbe non essere necessariamente un errore, qualora il programmatore avesse voluto che le istruzioni del `case 2:` fossero eseguite ogni volta che fosse stato eseguito il `case 1:`.

d) Errore: è stato utilizzato un operatore relazionale inappropriato nella condizione di continuazione dell'iterazione.
Correzione: utilizzate <= invece di <.

Esercizi

4.5 Trovate l'errore in ognuna delle seguenti istruzioni (*Nota*: ci potrebbe essere più di un errore):

a)
```
For ( x = 100, x >= 1, x++ )
printf( "%d\n", x );
```

b) Il seguente codice dovrebbe visualizzare se un dato intero è pari o dispari:

```
switch ( value % 2 ) {
case 0:
   printf( "Even integer\n" );
case 1:
   printf( "Odd integer\n" );
}
```

c) Il codice seguente dovrebbe prendere in input un intero e un carattere e visualizzarli. Supponete che l'utente immetta come input `100 A`.

```
scanf( "%d", &intVal );
charVal = getchar();
printf( "Integer: %d\nCharacter: %c\n", intVal, charVal );
```

d)
```
for ( x = .000001; x <= .0001; x += .000001 )
   printf( "%.7f\n", x );
```

e) Il codice seguente dovrebbe inviare in output gli interi dispari da 999 a 1:
```
for ( x = 999; x >= 1; x +=2 )
   printf( "%d\n", x );
```

f) Il codice seguente dovrebbe visualizzare gli interi pari da 2 a 100:
```
counter = 2;
```

```
Do {
   if ( counter % 2 == 0 )
   printf( "%d\n", counter );

   counter += 2;
} While ( counter < 100 );
```

g) Il codice seguente dovrebbe sommare gli interi da 100 a 150 (supponete che `total` sia già stata inizializzata a 0):

```
for ( x = 100; x <= 150; x++ );
   total += x;
```

4.6 Stabilite quali valori della variabile di controllo `x` saranno visualizzati da ognuna delle seguenti istruzioni `for`:

a)
```
for( x = 2; x <= 13; x += 2 )
   printf( "%d\n", x );
```

b)
```
for( x = 5; x <= 22; x += 7 )
   printf( "%d\n", x );
```

c)
```
for( x = 3; x <= 15; x += 3 )
   printf( "%d\n", x );
```

d)
```
for( x = 1; x <= 5; x += 7 )
   printf( "%d\n", x );
```

e)
```
for( x = 12; x >= 2; x -= 3 )
   printf( "%d\n", x );
```

4.7 Scrivete delle istruzioni `for` che visualizzino le seguenti sequenze di valori:

a) 1, 2, 3, 4, 5, 6, 7
b) 3, 8, 13, 18, 23
c) 20, 14, 8, 2, -4, -10
d) 19, 27, 35, 43, 51

4.8 Che cosa farà il programma seguente?

```
1  #include <stdio.h>
2
3  /* l'esecuzione del programma inizia dalla funzione main */
4  int main()
5  {
6     int x;
7     int y;
8     int i;
9     int j;
10
11    /* sollecita l'utente a inserire i dati */
12    printf( "Enter integers in the range 1-20: " );
13    scanf( "%d%d", &x, &y ); /* legge i valori per x e y */
14
15    for ( i = 1; i <= y; i++ ) { /* conta da 1 a y */
16
17       for ( j = 1; j <= x; j++ ) { /* conta da 1 a x */
18          printf( "@" ); /* visualizza @ */
19       } /* fine del comando for interno */
20
21       printf( "\n" ); /* inizia una nuova linea */
22    } /* fine del comando for esterno */
```

```
23
24      return 0; /* indica che il programma è terminato con successo */
25
26   } /* fine della funzione main */
```

4.9 Scrivete un programma che sommi una sequenza di interi. Supponete che il primo intero letto con la `scanf` specifichi il numero dei valori che dovranno essere immessi. Il vostro programma dovrà leggere solo un valore per ogni volta che la `scanf` sarà eseguita. Una tipica sequenza di input potrebbe essere:

```
5 100 200 300 400 500
```

dove **5** indica che dovranno essere sommati i cinque valori successivi.

4.10 Scrivete un programma che calcoli e visualizzi la media di diversi interi. Supponete che l'ultimo valore letto con la `scanf` sia quello della sentinella `9999`. Una tipica sequenza di input potrebbe essere:

```
10 8 11 7 9 9999
```

che indica che dovrà essere calcolata la media di tutti i valori che precedono `9999`.

4.11 Scrivete un programma che trovi il minore di diversi interi. Supponete che il primo valore letto specifichi il numero di quelli ancora da leggere.

4.12 Scrivete un programma che calcoli e visualizzi la somma degli interi pari da 2 a 30.

4.13 Scrivete un programma che calcoli e visualizzi il prodotto degli interi dispari da 1 a 15.

4.14 La funzione fattoriale è utilizzata frequentemente nei problemi di probabilità. Il fattoriale di un intero positivo *n*, scritto n! e pronunciato "fattoriale di *n*", è uguale al prodotto degli interi positivi da 1 a *n*. Scrivete un programma che valuti i fattoriali degli interi da 1 a 5. Visualizzate i risultati in un formato tabulare. Quale difficoltà potrà impedirvi di calcolare il fattoriale di 20?

4.15 Modificate il programma dell'interesse composto della Sezione 4.6 in modo che ripeta i suoi passi per tassi d'interesse del 5, 6, 7, 8, 9 e 10 per cento. Utilizzate un ciclo `for` per variare il tasso di interesse.

4.16 Scrivete un programma che visualizzi separatamente l'uno sotto l'altro i seguenti disegni. Utilizzate dei cicli `for` per generare i disegni. Tutti gli asterischi `(*)` dovranno essere stampati da una singola istruzione `printf` della forma `printf("*");` (ciò causerà la visualizzazione fianco a fianco degli asterischi). *Suggerimento*: gli ultimi due disegni richiederanno che ogni riga incominci con un numero appropriato di spazi.

```
(A)          (B)          (C)          (D)
*            **********   **********         *
**           *********    *********         **
***          ********     ********         ***
****         *******      *******         ****
*****        ******       ******         *****
******       *****        *****         ******
*******      ****         ****         *******
********     ***          ***         ********
*********    **           **         *********
**********   *            *         **********
```

4.17 Riscuotere il denaro diventa sempre più difficile nei periodi di recessione, per questo motivo le aziende potrebbero restringere i loro limiti di credito, per evitare che i propri conti di credito (il denaro

loro dovuto) diventino troppo sostanziosi. Come risposta a una recessione prolungata, una azienda ha dimezzato il limite di credito dei propri clienti. Di conseguenza, se un particolare cliente avesse avuto un limite di credito di $ 2000, questo sarebbe ora di $ 1000. Se un cliente avesse avuto un limite di credito di $ 5000, questo sarebbe ora di $ 2500. Scrivete un programma che analizzi lo stato di credito per tre clienti di questa azienda. Per ogni cliente vi saranno forniti:

1. Il numero di conto del cliente
2. Il limite di credito del cliente prima della recessione
3. Il saldo corrente del cliente (vale a dire, l'ammontare che il cliente deve all'azienda).

Il vostro programma dovrà calcolare e visualizzare il nuovo limite di credito, per ogni cliente, e dovrà anche determinare (e visualizzare) quali di loro hanno dei saldi correnti che eccedono il loro nuovo limite di credito.

4.18 Una interessante applicazione dei computer è il disegno di diagrammi e di grafici a barre (detti a volte "istogrammi"). Scrivete un programma che legga cinque numeri (ognuno compreso tra 1 e 30). Per ogni numero letto, il vostro programma dovrà visualizzare una riga contenente quel numero di asterischi adiacenti. Per esempio, se il vostro programma leggesse il numero sette, dovrà visualizzare *******.

4.19 Una azienda di vendita per corrispondenza vende cinque differenti prodotti i cui prezzi al dettaglio sono mostrati nella seguente tabella:

Numero di prodotto	Prezzo al dettaglio
1	2.98
2	4.50
3	9.98
4	4.49
5	6.87

Scrivete un programma che legga una serie di coppie di numeri come segue:

1) Numero di prodotto
2) Quantità venduta in un giorno

Il vostro programma dovrà utilizzare una istruzione `switch` per aiutare a determinare il prezzo al dettaglio di ogni prodotto. Il vostro programma dovrà calcolare e visualizzare il valore totale al dettaglio di tutti i prodotti venduti nell'ultima settimana.

4.20 Completate le seguenti tabelle di verità, riempiendo i relativi spazi con 0 o 1.

Condizione1	Condizione2	Condizione1 && Condizione2
0	0	0
0	diverso da zero	0
diverso da zero	0	_____
diverso da zero	diverso da zero	_____

Condizione1	Condizione2	Condizione1 || Condizione2
0	0	0
0	diverso da zero	1
diverso da zero	0	_____
diverso da zero	diverso da zero	_____

Condizione1	!Condizione1
0	1
diverso da zero	_____

4.21 Riscrivete il programma della Figura 4.2 in modo che l'inizializzazione della variabile `counter` sia eseguita nella dichiarazione, invece che nel comando `for`.

4.22 Modificate il programma della Figura 4.7 in modo che calcoli la votazione media della classe.

4.23 Modificate il programma della Figura 4.6 in modo che utilizzi soltanto degli interi per calcolare l'interesse composto. [*Suggerimento*: trattate tutte le cifre monetarie come quantità intere di centesimi. In seguito, "separate" il risultato nelle sue porzioni di dollari e centesimi, utilizzando rispettivamente le operazioni di divisione e modulo. Inserite un punto.]

4.24 Supponete che `i = 1, j = 2, k = 3` e `m = 2`. Che cosa visualizzerà ognuna delle seguenti istruzioni?

a) `printf( "%d", i == 1 );`
b) `printf( "%d", j == 3 );`
c) `printf( "%d", i >= 1 && j < 4 );`
d) `printf( "%d", m <= 99 && k < m );`
e) `printf( "%d", j >= i || k == m );`
f) `printf( "%d", k + m < j || 3 - j >= k );`
g) `printf( "%d", !m );`
h) `printf( "%d", !( j - m ) );`
i) `printf( "%d", !( k > m ) );`
j) `printf( "%d", !( j > k ) );`

4.25 Visualizzate una tabella di equivalenza tra decimali, binari, ottali ed esadecimali. Qualora non abbiate familiarità con i suddetti sistemi numerici, consultate prima l'Appendice E, se volete tentare di eseguire questo esercizio.

4.26 Calcolate il valore di π a partire dalla serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Visualizzate una tabella che mostri il valore di π approssimato da uno, due, tre ... termini di questa serie. Quanti termini di questa serie dovrete utilizzare, prima di ottenere 3,14? 3,141? 3,1415? 3,14159?

4.27 (*Terne pitagoriche*) Un triangolo rettangolo può avere lati che siano tutti interi. L'insieme dei tre valori interi per i lati di un triangolo rettangolo è detto terna pitagorica. Questi tre lati

devono soddisfare la relazione secondo la quale la somma dei quadrati dei due cateti è uguale al quadrato della ipotenusa. Trovate tutte le terne pitagoriche per lato1, lato2 e ipotenusa che non siano maggiori di 500. Utilizzate un triplo ciclo `for` nidificato che provi semplicemente tutte le possibilità. Questo è un esempio di elaborazione con "forza bruta". Non piace a molti dal punto di vista estetico. Esistono però molte ragioni per cui queste tecniche sono importanti. In primo luogo, con l'incremento della potenza di calcolo a un ritmo così fenomenale, le soluzioni che avrebbero richiesto anni o addirittura secoli di elaborazione, per essere ottenute con la tecnologia di solo un po' di anni fa, oggigiorno potrebbero essere ottenute in ore, in minuti o addirittura in secondi! In secondo luogo, come apprenderete nei corsi più avanzati di informatica, c'è un gran numero di problemi interessanti per i quali non esiste nessun approccio algoritmico noto diverso da quello della forza bruta. In questo libro investigheremo su molti tipi di metodologie per la soluzione dei problemi. Considereremo molti approcci con forza bruta a diversi problemi interessanti.

4.28 Una azienda retribuisce i propri dipendenti distinguendoli tra manager (che ricevono un salario fisso settimanale), lavoratori a ore (che ricevono una paga oraria fissa per le prime 40 ore di lavoro e "un'ora e mezza", ovverosia 1,5 volte la paga oraria, per le ore di straordinario effettuate), lavoratori a provvigione (che ricevono $250 più il 5,7% delle loro vendite lorde settimanali), oppure lavoratori a cottimo (che ricevono una cifra fissa di denaro ad articolo, per ognuno di quelli prodotti: in questa azienda ogni lavoratore a cottimo lavora soltanto su un tipo di articolo). Scrivete un programma che calcoli la paga settimanale di ogni dipendente. Non conoscete in anticipo il numero dei dipendenti. Ogni tipo di dipendente ha il proprio codice di pagamento: i manager hanno 1, i lavoratori a ore hanno 2, quelli a provvigione hanno 3 e quelli a cottimo hanno 4. Utilizzate un comando `switch` per calcolare la paga di ogni dipendente in base al suo codice di pagamento. All'interno del comando `switch`, richiedete all'utente (ovverosia all'impiegato addetto alle paghe) di immettere i fatti appropriati di cui il vostro programma avrà bisogno per calcolare la paga di ogni dipendente, in base al proprio codice di pagamento.

4.29 (*Leggi di De Morgan*) In questo capitolo abbiamo discusso degli operatori logici `&&`, `||` e `!`. Le Leggi di De Morgan potranno a volte renderci più pratica la formulazione di una espressione logica. Queste leggi stabiliscono che l'espressione `!(condizione1 && condizione2)` è logicamente equivalente alla espressione `(!condizione1 || !condizione2)`. Inoltre, l'espressione `!(condizione1 || condizione2)` è logicamente equivalente alla espressione `(!condizione1 && !condizione2)`. Utilizzate le Leggi di De Morgan per scrivere delle espressioni equivalenti per ognuna di quelle seguenti e quindi scrivete un programma che dimostri che l'espressione originale e quella nuova sono, in ogni caso, equivalenti:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( Y > 4 ) )`
d) `!( (i > 4 ) || ( j <= 6 ) )`

4.30 Riscrivete il programma della Figura 4.7 sostituendo l'istruzione `switch` con dei comandi `if...else` nidificati; state attenti a trattare appropriatamente il caso di `default`. Riscrivete quindi questa nuova versione, sostituendo i comandi `if...else` nidificati con una serie di istruzioni `if`; anche questa volta, state attenti a trattare appropriatamente il caso di `default` (ciò sarà più difficile che con la versione dei comandi `if...else` nidificati). Questo esercizio dimostrerà che il comando `switch` è una comodità e che ogni `switch` potrà essere sostituito con istruzioni di selezione singola.

4.31 Scrivete un programma che visualizzi il seguente disegno di un rombo. Potrete utilizzare delle istruzioni `printf` che visualizzino ognuna un singolo asterisco (`*`) o uno spazio. Massimizzate il vostro utilizzo delle ripetizioni (con comandi `for` nidificati) e minimizzate il numero di istruzioni `printf`.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

4.32 Modificate il programma che avete scritto nell'Esercizio 4.31, in modo da leggere un numero dispari compreso nell'intervallo da 1 a 19 per specificare il numero di righe comprese nel rombo. Il vostro programma dovrà quindi visualizzare un rombo della misura appropriata.

4.33 Scrivete un programma che visualizzi una tabella contenente tutti i numeri romani equivalenti a quelli decimali compresi nell'intervallo da 1 a 100.

4.34 Scrivete un programma che visualizzi una tabella contenente i numeri binari, ottali ed esadecimali equivalenti a quelli decimali compresi nell'intervallo da 1 a 256. Se non avete familiarità con questi sistemi numerici, consultate prima l'Appendice E, se volete tentare di eseguire questo esercizio.

4.35 Descrivete il processo che usereste per sostituire un ciclo `do...while` con un comando `while` equivalente. Quale problema incontrereste qualora tentaste di sostituire un ciclo `while` con un comando `do...while` equivalente? Supponete che vi sia stato chiesto di rimuovere un ciclo `while` e di sostituirlo con un comando `do...while`. Di quale altro comando di controllo avrete bisogno e come lo userete, per assicurarvi che il programma risultante si comporti esattamente come l'originale?

4.36 Scrivete un programma che prenda in input un anno compreso nell'intervallo dal 1994 al 1999 e utilizzi una iterazione con un ciclo **`for`** per produrre un calendario compendiato, visualizzato in modo ordinato. Attenti agli anni bisestili.

4.37 Una critica alle istruzioni `break` e `continue` è che non sono strutturate. In realtà le istruzioni `break` e `continue` possono sempre essere sostituite con istruzioni strutturate, sebbene farlo possa essere scomodo. Descrivete in generale come rimuovereste tutte le istruzioni `break` dal ciclo di un programma, e come le sostituireste con qualche istruzione strutturata equivalente. [*Suggerimento*: l'istruzione `break` abbandona un ciclo dall'interno del suo corpo. L'altra maniera per abbandonarlo è facendo in modo che il controllo di continuazione del ciclo fallisca. Nel controllo di continuazione del ciclo, considerate l'utilizzo di un secondo controllo che indichi "uscita anticipata a causa di una condizione di 'break'".] Utilizzate la tecnica che avete appreso in questo esercizio per rimuovere l'istruzione di interruzione dal programma della Figura 4.11.

4.38 Che cosa farà il seguente frammento di programma?

```
1  for ( i = 1; i <= 5; i++ ) {
2     for ( j = 1; j <= 3; j++ ) {
3        for ( k = 1; k <= 4; k++ )
4           printf( "*" );
5        printf( "\n" );
6     }
7     printf( "\n" );
8  }
```

4.39 Descrivete in generale come rimuovereste tutte le istruzioni `continue` dal ciclo di un programma, e come le sostituireste con qualche istruzione strutturata equivalente. Utilizzate la tecnica che avete sviluppato in questo esercizio per rimuovere l'istruzione `continue` dal programma della Figura 4.12

Capitolo 5

Le funzioni in C

Obiettivi

- Comprendere come costruire i programmi in modo modulare partendo da piccoli pezzi chiamati funzioni.
- Introdurre le comuni funzioni matematiche disponibili nella libreria standard del C.
- Essere in grado di creare nuove funzioni.
- Comprendere i meccanismi utilizzati per passare le informazioni alle funzioni.
- Introdurre le tecniche di simulazione che utilizzano la generazione di numeri casuali.
- Comprendere come scrivere e utilizzare le funzioni che richiamano se stesse.

5.1 Introduzione

La maggior parte dei programmi per computer che siano stati scritti per risolvere problemi del mondo reale sono più corposi dei programmi presentati sinora, in questi primi capitoli. L'esperienza ha dimostrato che il modo migliore, per sviluppare e amministrare un programma corposo, è di costruirlo partendo da pezzi più piccoli o *moduli* ognuno dei quali sia più maneggevole del programma originale. Questa tecnica è detta *dividi e conquista* (dal latino *divide et impera*). Questo capitolo descriverà le caratteristiche del linguaggio C che facilitano la progettazione, l'implementazione, il funzionamento e la manutenzione di programmi corposi.

5.2 I moduli di programma in C

I moduli in C sono chiamati *funzioni*. I programmi C sono scritti tipicamente combinando le nuove funzioni scritte dal programmatore con quelle "preconfezionate" disponibili nella *libreria standard del C*. In questo capitolo discuteremo di entrambi i tipi di funzione. La libreria standard del C fornisce una ricca collezione di funzioni per eseguire i comuni calcoli matematici, per la manipolazione delle stringhe e dei caratteri, per l'input/output e per molte altre operazioni utili. Ciò renderà più semplice il lavoro del programmatore, poiché le suddette funzioni forniranno molte delle capacità di cui egli avrà bisogno.

Buona abitudine 5.1

Familiarizzate con la ricca collezione di funzioni incluse nella libreria standard del C.

Ingegneria del software 5.1

Evitate di "inventare nuovamente la ruota". Utilizzate le funzioni incluse nella libreria standard del C, qualora sia possibile, invece di scrivere delle nuove funzioni. Ciò ridurrà il tempo di sviluppo del programma.

Obiettivo portabilità 5.1

Utilizzare le funzioni della libreria standard del C aiuterà a rendere più portabili i programmi.

Per quanto le funzioni della libreria standard non facciano tecnicamente parte del linguaggio C, esse sono fornite immancabilmente con tutti i sistemi C. Le funzioni `printf`, `scanf` e `pow` che abbiamo utilizzato nei capitoli precedenti sono appunto delle funzioni della libreria standard.

Per definire dei compiti specifici, il programmatore potrà scrivere delle funzioni che possano essere utilizzate in molti punti del programma. Queste sono a volte chiamate *funzioni definite dal programmatore*. Le istruzioni che definiscono effettivamente la funzione, saranno scritte solo una volta e saranno nascoste alle altre funzioni.

Le funzioni sono *invocate* da una *chiamata di funzione* che specifica il nome della funzione e fornisce delle informazioni (gli *argomenti*) di cui la funzione chiamata ha bisogno, per completare le attività per le quali è stata progettata. Una tipica analogia per tutto ciò è quella della struttura gerarchica di un'azienda. Un capo (la *funzione chiamante* o *chiamante*) chiede a un operaio (la *funzione chiamata*) di eseguire un compito e tornare indietro a riferire, quando il lavoro sarà stato eseguito (Figura 5.1). Per esempio, una funzione che voglia visualizzare delle informazioni sullo schermo richiamerà la funzione operaia `printf` per eseguire quel compito, in seguito `printf` visualizzerà le suddette informazioni e tornerà indietro a riferire, ovverosia *restituirà il controllo del programma*, alla funzione chiamante quando il suo compito sarà stato completato. La funzione capo non sa in che modo quell'operaia eseguirà i compiti che le sono stati assegnati. L'operaia potrebbe anche richiamare altre funzioni operaie e il capo non ne sarebbe a conoscenza. Vedremo presto in che modo questo "incapsulamento" dei dettagli dell'implementazione promuoverà una buona progettazione del software. La Figura 5.1 mostra la funzione `main` mentre comunica con diverse operaie in modo gerarchico. Osservate che `worker1` agisce come una funzione capo per `worker4` e `worker5`. Le relazioni tra le funzioni potrebbero anche essere diverse dalla struttura gerarchica mostrata in questa figura.

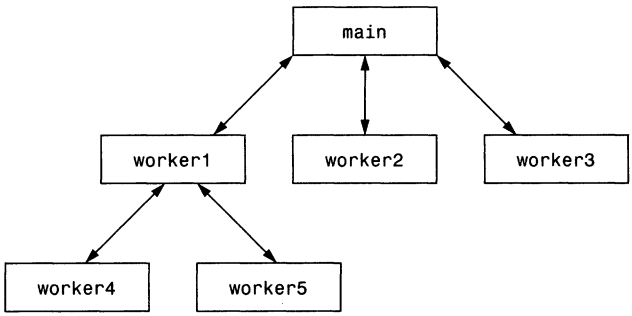


Figura 5.1 Relazione gerarchica tra la funzione capo e quelle operaie

5.3 Le funzioni della libreria matematica

Le funzioni della libreria matematica consentono al programmatore di eseguire certi tipici calcoli matematici. In questo contesto utilizzeremo varie funzioni matematiche per introdurre il concetto di funzione. Più tardi nel libro, discuteremo molte delle altre funzioni incluse nella libreria standard del C.

Normalmente le funzioni sono utilizzate in un programma scrivendo il nome delle stesse, seguito da una parentesi tonda aperta, dall'*argomento* (oppure da una lista di argomenti separati da virgole) della funzione e da una parentesi tonda chiusa. Per esempio, un programmatore che desideri calcolare e visualizzare la radice quadrata di `900,0` potrà scrivere:

```
printf( "%.2f", sqrt( 900.0 ) );
```

Nel momento in cui questa istruzione sarà eseguita, sarà invocata la funzione `sqrt` della libreria matematica per calcolare la radice quadrata del numero contenuto nelle parentesi (`900.0`). Il numero `900,0` è l'argomento della funzione `sqrt`. L'istruzione precedente visualizzerà `30,00`. La funzione `sqrt` accetta un argomento di tipo `double` e restituisce un risultato `double`. Tutte le funzioni della libreria matematica restituiscono il tipo di dato `double`. Si osservi che i valori di tipo `double`, come i valori di tipo `float`, possono essere mandati in output usando la specifica di conversione `%f`.

Collaudo e messa a punto 5.1

Includete il file di intestazione matematico, utilizzando la direttiva del preprocessore `#include <math.h>`*, quando utilizzate le funzioni della libreria matematica.*

Gli argomenti della funzione possono essere delle costanti, variabili o espressioni. Supponendo che `c1 = 13,0`, `d = 3,0` e `f = 4,0`, l'istruzione

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

calcolerà e visualizzerà la radice quadrata di `13,0 + 3,0 * 4,0 = 25,0`, vale a dire `5,00`.

Nella Figura 5.2 sono riassunte alcune delle funzioni incluse nella libreria matematica del C. Nella figura le variabili `x` e `y` sono di tipo `double`.

5.4 Le funzioni

Le funzioni consentono al programmatore di suddividere in moduli un programma. Tutte le variabili dichiarate nelle definizioni di funzione sono *variabili locali*: esse sono note soltanto in quella in cui sono state definite. La maggior parte delle funzioni contiene una lista di *parametri*. Questi forniscono il mezzo per comunicare le informazioni tra le funzioni. Anche i parametri di una funzione sono delle variabili locali di quest'ultima.

Ingegneria del software 5.2

Nei programmi contenenti molte funzioni, `main` *potrebbe essere implementato come un gruppo di chiamate a funzioni che eseguono il grosso del lavoro del programma.*

Funzione	Descrizione	Esempio
`sqrt(x)`	radice quadrata di *x*	`sqrt(900.0)` è `30,0` `sqrt(9.0)` è `3,0`
`exp(x)`	funzione esponenziale *e*	`exp(1.0)` è `2,718282` `exp(2.0)` è `7,389056`
`log(x)`	logaritmo naturale di *x* (in base *e*)	`log(2.718282)` è `1,0` `log(7.389056)` è `2,0`

`log10(x)`	logaritmo di *x* (in base 10)	`log10(1.0)` è `0,0`
		`log10(10.0)` è `1,0`
		`log10(100.0)` è `2,0`

`fabs(x)`	valore assoluto di *x*	`fabs(5.0)` è `5.0`
		`fabs(0.0)` è `0,0`
		`fabs(-5.0)` è `-5,0`

`ceil(x)`	arrotonda *x* all'intero più piccolo non minore di *x*	`ceil(9.2)` è `10,0` `ceil(-9.8)` è `-9,0`
`floor(x)`	arrotonda *x* all'intero più grande non maggiore di *x*	`floor(9.2)` è `9,0` `floor(-9.8)` è `-10,0`
`pow(x, y)`	*x* elevato alla potenza *y* ()	`pow(2, 7)` è `128,0` `pow(9, .5)` è `3,0`
`fmod(x, y)`	resto di *x*/*y* in virgola mobile	`fmod(13.657, 2.333)` è `1,992`
`sin(x)`	seno trigonometrico di *x* (*x* è espressa in radianti)	`sin(0.0)` è `0,0`
`cos(x)`	coseno trigonometrico di *x* (*x* è espressa in radianti)	`cos(0.0)` è `1,0`
`tan(x)`	tangente trigonometrica di *x* (*x* è espressa in radianti)	`tan(0.0)` è `0,0`

Figura 5.2 Funzioni comunemente utilizzate della libreria matematica

Ci sono diverse motivazioni per "suddividere in funzioni" un programma. L'approccio dividi e conquista rende maneggevole lo sviluppo del programma. Un'altra motivazione è la *riusabilità del software*: utilizzare le funzioni esistenti come blocchi di costruzione per creare i nuovi programmi. La riusabilità del software è uno degli elementi principali del movimento per la programmazione orientata agli oggetti, che imparerete quando studierete linguaggi derivati dal C come il C++, Java e il C# (da pronunciare "C sharp"). Con un buon nome di funzione e una buona definizione, i programmi potranno essere creati da funzioni standardizzate che eseguano dei compiti specifici, piuttosto che essere costruiti utilizzando un codice personalizzato. Questa tecnica è conosciuta come *astrazione*. Noi usiamo l'astrazione ogni volta che scriviamo dei programmi che includano delle funzioni incluse nella libreria standard, come `printf`, `scanf` e `pow`. Una terza motivazione è di evitare la ripetizione del codice all'interno di un programma. Impacchettare il codice in forma di funzione consentirà allo stesso di essere eseguito in diversi punti del programma richiamando semplicemente la funzione.

Ingegneria del software 5.3

Ogni funzione dovrebbe limitarsi a eseguire un compito singolo e ben definito, mentre il nome della funzione dovrebbe esprimere efficacemente quel compito. Ciò faciliterà l'astrazione e promuoverà la riusabilità del software.

Ingegneria del software 5.4

Se non riuscite a scegliere un nome conciso che esprima l'attività svolta dalla vostra funzione, è probabile che questa stia tentando di eseguire troppi compiti diversi. In questi casi, di solito è meglio suddividerla in diverse funzioni più piccole.

5.5 Le definizioni di funzione

Ognuno dei programmi che abbiamo presentato è stato formato da una funzione chiamata `main` che, per eseguire i propri compiti, ha richiamato quelle della libreria standard. Consideriamo ora in che modo i programmatori possano scrivere le proprie funzioni personalizzate.

Considerate un programma che utilizzi una funzione `square` per calcolare e visualizzare i quadrati degli interi compresi tra 1 e 10 (Figura 5.3).

```
1  /* Fig. 5.3: fig05_03.c
2     Creazione e utilizzo di una funzione definita dal programmatore */
3  #include <stdio.h>
4
5  int square( int y );   /* prototipo della funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    int x; /* contatore */
11
12    /* itera 10 volte e calcola e visualizza il quadrato di x
         a ogni ciclo */
13    for ( x = 1; x <= 10; x++ ) {
14       printf( "%d  ", square( x ) ); /* chiamata di funzione */
15    }
16
17    printf( "\n" );
18
19    return 0; /* indica che il programma è terminato con successo */
20
21 } /* fine della funzione main */
22
23 /* La definizione della funzione square restituisce il quadrato
      del parametro */
24 int square( int y ) /* y è una copia dell'argomento passato
                          alla funzione */
25 {
```

Figura 5.3 Usare una funzione definita dal programmatore (continua)

```
26      return y * y; /* restituisce il quadrato di y come intero */
27
28   } /* fine della funzione square */
```

```
1  4  9  16  25  36  49  64  81  100
```

Figura 5.3 Usare una funzione definita dal programmatore

Buona abitudine 5.2

Inserite una riga vuota tra le definizioni di funzione, per separarle e aumentare la leggibilità del programma.

La funzione `square` sarà *invocata* o *richiamata* nel corpo di `main` all'interno dell'istruzione `printf` (riga 14)

```
printf( "%d  ", square( x ) ); /* chiamata di funzione */
```

La funzione `square` riceverà una copia del valore di `x` nel *parametro* `y` (riga 24). In seguito `square` calcolerà `y * y` (riga 26). Il risultato sarà restituito alla funzione `printf` all'interno del `main` nel punto dell'invocazione di `square` e `printf` visualizzerà il risultato. Questo processo sarà ripetuto 10 volte utilizzando il comando di iterazione `for`.

La definizione della funzione `square` mostra che questa attenderà il parametro intero `y`. La parola chiave `int` che precede il nome della funzione (riga 24) indica che `square` restituirà un risultato intero. L'istruzione `return` nella funzione `square` restituirà il risultato del calcolo alla funzione chiamante.

La riga 5

```
int square( int y );   /* prototipo della funzione */
```

è un *prototipo di funzione*. L'`int` all'interno delle parentesi informa il compilatore che `square` si aspetterà di ricevere un valore intero dal chiamante. L'`int` alla sinistra del nome della funzione `square` informa il compilatore che questa restituirà al chiamante un risultato intero. Il compilatore farà riferimento al prototipo della funzione, per controllare che le chiamate a `square` (riga 14) contengano il tipo di ritorno corretto, il numero e i tipi appropriati per gli argomenti e che questi siano forniti nell'ordine corretto. I prototipi di funzione saranno trattati in dettaglio nella Sezione 5.6.

Il formato di una definizione di funzione è

tipo-del-valore-di-ritorno nome-della-funzione(*lista-dei-parametri*)

{

dichiarazioni

istruzioni

}

Il *nome della funzione* è un qualsiasi identificatore valido. Il *tipo del valore di ritorno* è quello del risultato restituito al chiamante. Il *tipo del valore di ritorno* `void` indica che una funzione non restituirà alcun valore. Un *tipo del valore di ritorno* non specificato sarà sempre conside-

rato un `int` dal compilatore. Tuttavia, l'omissione del tipo del valore di ritorno è una pratica da non incentivare. Il *tipo del valore di ritorno*, il *nome-della-funzione* e la *lista-dei-parametri*, presi assieme, sono talvolta denominati l'*intestazione della funzione*.

Errore tipico 5.1

Omettere il tipo per il valore di ritorno in una definizione di funzione è un errore di sintassi, qualora il prototipo della funzione specifichi per il valore di ritorno un tipo diverso da `int`.

Errore tipico 5.2

Dimenticare di restituire un valore da una funzione che dovrebbe farlo potrà condurre a errori inattesi. Lo standard C stabilisce che il risultato di questo tipo di omissioni sarà indefinito.

Errore tipico 5.3

Restituire un dato da una funzione con tipo del valore di ritorno `void` *è un errore di sintassi.*

Buona abitudine 5.3

Stabilite sempre in modo esplicito il tipo di dato restituito, anche se la sua omissione farebbe restituire un `int` *per default.*

La *lista dei parametri* è un elenco che specifica i parametri, separati da virgole, che saranno ricevuti dalla funzione quando sarà richiamata. Qualora una funzione non riceva alcun valore, la *lista dei parametri* sarà `void`. Il tipo di ognuno dei parametri dovrà essere specificato esplicitamente, salvo che non siano di tipo `int`. Infatti, qualora non sia stato specificato il compilatore presumerà che si tratti di un `int`.

Errore tipico 5.4

Dichiarare i parametri della funzione, qualora siano dello stesso tipo, usando la forma `double x, y` *invece che* `double x, double y` *potrebbe provocare degli errori nei vostri programmi. La dichiarazione* `double x, y` *in realtà renderebbe* `y` *un parametro di tipo* `int`, *poiché* `int` *è il default.*

Errore tipico 5.5

È un errore di sintassi inserire un punto e virgola, dopo la parentesi destra che chiude l'elenco dei parametri in una definizione di funzione.

Errore tipico 5.6

È un errore di sintassi ridefinire un parametro della funzione come variabile locale alla stessa.

Buona abitudine 5.4

Includere il tipo di dato per ogni parametro presente nel relativo elenco, anche qualora fosse del tipo di default `int`.

Buona abitudine 5.5

Per quanto non sia scorretto farlo, non utilizzate gli stessi nomi per gli argomenti passati a una funzione e per i corrispondenti parametri inseriti nella relativa definizione. Ciò aiuterà a evitare ambiguità.

Le *dichiarazioni* e le *istruzioni* inserite all'interno delle parentesi graffe formano il *corpo della funzione*. Il corpo della funzione è chiamato anche *blocco*. Le variabili potranno essere dichiarate in qualsiasi blocco e questi potranno essere nidificati. *Una funzione non può mai essere definita all'interno di un'altra funzione.*

Errore tipico 5.7

Definire una funzione all'interno di un'altra funzione è un errore di sintassi.

Buona abitudine 5.6

Scegliere dei nomi di funzione e di parametro significativi renderà i programmi più leggibili e aiuterà a evitare l'uso eccessivo di commenti.

Ingegneria del software 5.5

Una funzione non dovrebbe essere più lunga di una pagina. Meglio ancora, una funzione non dovrebbe essere più lunga di una mezza pagina. Le funzioni piccole favoriscono la riusabilità del software.

Ingegneria del software 5.6

I programmi dovrebbero essere scritti come collezioni di funzioni piccole. Ciò li renderà più semplici da scrivere, collaudare, mettere a punto e modificare.

Ingegneria del software 5.7

Una funzione che richieda un gran numero di parametri potrebbe dover svolgere troppi compiti. Considerate la possibilità di suddividerla in funzioni più piccole che eseguano dei compiti distinti. L'intestazione della funzione dovrebbe rientrare in una riga, se possibile.

Ingegneria del software 5.8

Il prototipo, l'intestazione e la chiamata della funzione dovrebbero tutti concordare nel numero, nel tipo e nell'ordine degli argomenti e dei parametri oltre che nel tipo del valore restituito.

Esistono tre modi per restituire il controllo da una funzione chiamata al punto in cui quest'ultima è stata invocata. Nel caso in cui la funzione non restituisca alcun risultato, il controllo sarà restituito semplicemente quando sarà raggiunta la parentesi graffa destra che chiude la funzione, oppure eseguendo l'istruzione

```
return;
```

Nel caso che la funzione restituisca un risultato, l'istruzione

```
return espressione;
```

restituirà al chiamante il valore dell'*espressione*.

Il nostro secondo esempio utilizzerà la funzione `maximum` definita dall'utente, per determinare e restituire il maggiore fra tre interi (Figura 5.4). I tre interi saranno presi in input con `scanf` (riga 15). In seguito, gli interi saranno passati alla funzione `maximum` (riga 19) che determinerà quale sia il maggiore di essi. Questo valore sarà restituito al `main` dall'istruzione `return` della funzione `maximum` (riga 39). Il valore restituito sarà assegnato alla variabile `largest` che sarà visualizzata in seguito da `printf` (riga 19).

5.6 I prototipi di funzione

Una delle più importanti caratteristiche del C è il *prototipo di funzione*. Il comitato dello standard C prese in prestito questa caratteristica dagli sviluppatori del C++. Un prototipo di funzione indica al compilatore il tipo del dato restituito dalla funzione, il numero dei parametri che quella si aspetta di ricevere, il tipo dei parametri e l'ordine in cui questi sono attesi. Il compilatore utilizzerà i prototipi per convalidare le chiamate di funzione. Le versioni precedenti del C non eseguivano questo tipo di controllo, perciò era possibile richiamare le funzioni in modo improprio senza che il compilatore individuasse degli errori. Durante la fase di esecuzione, tali chiamate avrebbero potuto causare degli errori fatali, o anche non fatali, ma che avrebbero provocato degli errori logici subdoli e difficili da individuare. I prototipi di funzione pongono rimedio a questa lacuna.

```
1  /* Fig. 5.4: fig05_04.c
2     Trovare il maggiore di tre interi */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z );   /* prototipo della funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    int number1; /* primo intero */
11    int number2; /* secondo intero */
12    int number3; /* terzo intero */
13
14    printf( "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    /* number1, number2 e number3 sono gli argomenti
18       della chiamata di funzione a maximum */
19    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20
21    return 0; /* indica che il programma è terminato con successo */
22
23 } /* fine della funzione main */
24
25 /* Definizione della funzione maximum */
26 /* x, y e z sono parametri */
27 int maximum( int x, int y, int z )
28 {
```

Figura 5.4 La funzione maximum definita dal programmatore (continua)

```
29      int max = x; /* si assume che x sia il massimo */
30
31      if ( y > max ) { /* se y è maggiore di max, assegna y a max */
32         max = y;
33      } /* fine del comando if */
34
35      if ( z > max ) { /* se z è maggiore di max, assegna z a max */
36         max = z;
37      } /* fine del comando if */
38
39      return max; /* max è il valore massimo */
40
41   } /* fine della funzione maximum */
```

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 85 22 17
Maximum is: 85

Enter three integers: 22 17 85
Maximum is: 85
```

Figura 5.4 La funzione maximum definita dal programmatore

Buona abitudine 5.7

Includete per tutte le funzioni i relativi prototipi in modo da trarre vantaggio dalle capacità di controllo di tipo implementate nel linguaggio C. Utilizzate le direttive `#include` *del preprocessore per ottenere, dai file di intestazione appropriati, i prototipi delle funzioni incluse nella libreria standard o per ottenere le intestazioni contenenti i prototipi di funzione per le funzioni sviluppate da voi e dai membri del vostro gruppo.*

Il prototipo di funzione per la `maximum` della Figura 5.4 (riga 5) è

```
int maximum( int x, int y, int z );   /* prototipo della funzione */
```

Questo prototipo di funzione stabilisce che `maximum` riceverà tre argomenti `int` e restituirà un risultato dello stesso tipo. Osservate che il prototipo della funzione `maximum` è uguale alla prima riga della sua definizione.

Buona abitudine 5.8

A volte, i nomi dei parametri saranno inclusi nei prototipi di funzione (una nostra preferenza) per scopi documentativi. In ogni modo il compilatore li ignorerà.

Errore tipico 5.8

Dimenticare il punto e virgola alla fine del prototipo di funzione è un errore di sintassi.

Una chiamata di funzione che non corrisponda al suo prototipo è un errore di sintassi. Sarà generato un errore anche qualora il prototipo e la definizione della funzione non concordino. Per esempio, se il prototipo della funzione nella Figura 5.4 fosse stato scritto come

```
void maximum( int x, int y, int z );
```

il compilatore avrebbe generato un errore perché il tipo di ritorno `void`, indicato nel prototipo della funzione, sarebbe stato diverso da quello `int` specificato nell'intestazione della stessa.

Un'altra importante caratteristica dei prototipi di funzione è la *coercizione degli argomenti*, ovverosia la conversione forzata degli argomenti al tipo appropriato. Per esempio, la funzione della libreria matematica `sqrt` potrà essere richiamata con un argomento intero, funzionando ancora correttamente anche se il suo prototipo in `math.h` specifica che l'argomento debba essere di tipo `double`. L'istruzione

```
printf( "%.3f\n", sqrt( 4 ) );
```

valuterà correttamente `sqrt( 4 )` e visualizzerà il valore `2,000`. Il prototipo di funzione fa in modo che il compilatore converta il valore intero `4` in quello di tipo `double 4,0`, prima che lo stesso sia passato alla `sqrt`. In generale, i valori degli argomenti che non corrispondano precisamente ai tipi dei parametri definiti nel prototipo saranno convertiti in modo appropriato, prima che la funzione sia richiamata. Tali conversioni potrebbero però causare dei risultati scorretti, qualora non fossero rispettate le *regole di promozione* del C. Tali regole specificano in quale modo i vari tipi di dato possano essere convertiti tra loro senza perdita di informazioni. Nel nostro esempio con `sqrt`, un `int` sarà convertito automaticamente in un `double` senza che ne sia modificato il valore. Nondimeno un `double` convertito in `int` troncherebbe la parte frazionaria del valore `double`. Anche convertire i tipi più grandi degli interi in quelli più piccoli (per esempio, un `long` in uno `short`) potrà produrre dei valori modificati.

Le regole di promozione si applicano automaticamente alle espressioni che contengono dei valori di due o più tipi di dato, dette anche espressioni di *tipo misto*. Ogni valore in un'espressione di tipo misto sarà promosso automaticamente a quello "più alto" dell'espressione (in realtà, sarà creata e utilizzata una versione temporanea di ognuno dei valori: quelli originali rimarranno invariati). La Figura 5.5 elenca i tipi di dato nell'ordine da quello più alto a quello più basso, con le relative specifiche di conversione per `printf` e `scanf`.

Tipi di dato	Specifiche di conversione per printf	Specifiche di conversione per scanf
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Figura 5.5 Gerarchia di promozione per i tipi di dato

Convertire dei dati nei tipi più bassi produce normalmente dei valori scorretti. Di conseguenza, i valori potranno essere convertiti in un tipo più basso solo assegnandoli esplicitamente a una variabile di tipo più basso, o utilizzando l'operatore di conversione. I valori degli argomenti di funzione saranno convertiti ai tipi dei parametri definiti nel relativo prototipo, come se fossero stati assegnati direttamente a variabili di quei tipi. Qualora la nostra funzione `square` che utilizza un parametro intero (Figura 5.3) fosse stata richiamata con un argomento in virgola mobile, questo sarebbe stato convertito in un `int` (un tipo più basso) e `square` avrebbe restituito un valore scorretto. Per esempio, `square(4.5)` restituirebbe `16` invece di `20,25`.

Errore tipico 5.9

Convertire un tipo di dato più alto, rispetto alla gerarchia di promozione, in uno più basso potrà modificare il valore dei dati.

Il compilatore formerà un suo prototipo di funzione, qualora questo non sia stato incluso in un programma, utilizzando la prima occorrenza della funzione: ovverosia, la definizione o una sua chiamata. Per default, il compilatore assumerà che la funzione restituisca un `int` e non presumerà nulla relativamente agli argomenti. Ne consegue che, qualora gli argomenti passati alla funzione non siano corretti, gli errori non saranno individuati dal compilatore.

Errore tipico 5.10

Dimenticare un prototipo provocherà un errore di sintassi, qualora il tipo restituito dalla funzione non sia un `int` *e la definizione di quella appaia nel programma dopo una sua chiamata. In caso contrario, dimenticare il prototipo di una funzione potrà causare un errore durante la fase di esecuzione, oppure un risultato inatteso.*

Ingegneria del software 5.9

Un prototipo inserito all'esterno di ogni definizione di funzione si applicherà a tutte le relative invocazioni che appariranno nel file dopo il prototipo. Un prototipo inserito all'interno di una funzione si applicherà solo alle chiamate eseguite all'interno della stessa.

5.7 Lo stack delle chiamate di funzione e i record di attivazione

Per comprendere come il C effettui le chiamate di sistema, è necessario pensare ad una struttura dati (ovvero, una collezione di elementi informativi correlati) nota come *stack*. Possiamo pensare ad uno stack come a qualcosa di analogo ad una pila di piatti. Quando un piatto viene sistemato nella pila, esso viene normalmente posizionato in cima (questa attività viene descritta come il *push* di un piatto sullo stack). Similmente, quando un piatto viene rimosso dalla pila, la rimozione avviene sempre dalla cima (operazione descritta come il *pop* di un piatto dallo stack). Gli stack sono conosciuti come *strutture dati last-in, first-out* (LIFO), ovvero, l'ultimo elemento inserito (push) è anche il primo ad essere rimosso (pop).

Quando un programma richiama una funzione, la funzione chiamata deve sapere come restituire il controllo al chiamante; quindi l'indirizzo di ritorno della funzione chiamante viene inserito nello *stack di esecuzione del programma* (a volte chiamato *stack delle chiamate di*

funzione). Nel caso in cui si verifichi una sequenza di chiamate di funzione, gli indirizzi di ritorno successivi vengono inseriti sullo stack in ordine last-in, first-out in modo che ogni funzione possa restituire il controllo al proprio chiamante.

File di intestazione della libreria standard	Spiegazione
`<assert.h>`	Contiene le macro e le informazioni per aggiungere delle istruzioni diagnostiche che forniscono un aiuto durante la messa a punto del programma.
`<ctype.h>`	Contiene i prototipi per le funzioni che verificano talune proprietà dei caratteri, nonché per quelle che potranno essere utilizzate per convertire le lettere minuscole in maiuscole e viceversa.
`<errno.h>`	Definisce le macro che saranno utili per comunicare le condizioni di errore.
`<float.h>`	Contiene i limiti del sistema per la dimensione dei valori in virgola mobile.
`<limits.h>`	Contiene i limiti del sistema per la dimensione dei valori interi.
`<locale.h>`	Contiene i prototipi di funzione e altre informazioni che consentiranno a un programma di essere adattato alla località in cui sarà eseguito. La nozione di località consente al sistema di gestire le diverse convenzioni delle varie aree del mondo per esprimere informazioni come la data, l'ora, la valuta e i grandi numeri.
`<math.h>`	Contiene i prototipi per le funzioni della libreria matematica.
`<setjmp.h>`	Contiene i prototipi per le funzioni che consentono di aggirare l'usuale sequenza di chiamata e ritorno da funzione.
`<signal.h>`	Contiene i prototipi di funzione e le macro per gestire le varie condizioni che potranno insorgere durante l'esecuzione del programma.
`<stdarg.h>`	Definisce le macro che consentono di gestire delle funzioni per le quali il numero e i tipi degli argomenti siano sconosciuti.
`<stddef.h>`	Contiene le definizioni comuni dei tipi di dato utilizzati dal C per eseguire certi calcoli.
`<stdio.h>`	Contiene i prototipi per le funzioni di input/output della libreria standard e le informazioni utilizzate da queste ultime.
`<stdlib.h>`	Contiene i prototipi delle funzioni per la conversione dei numeri in testo e viceversa, per l'allocazione della memoria, per i numeri casuali e per altre funzioni di utilità generica.
`<string.h>`	Contiene i prototipi delle funzioni per l'elaborazione delle stringhe.
`<time.h>`	Contiene i prototipi di funzione e i tipi per la manipolazione dell'ora e delle date.

Figura 5.6 Alcuni file di intestazione della libreria standard

Lo stack di esecuzione del programma contiene anche la memoria per le variabili locali utilizzate ad ogni invocazione della funzione durante l'esecuzione del programma. Tali dati, memorizzati come una parte dello stack di esecuzione del programma, sono noti come *record di attivazione* o *frame dello stack* della chiamata di funzione. Quando viene effettuata una chiamata di funzione, il suo record di attivazione viene aggiunto allo stack di esecuzione del programma. Quando la funzione restituisce il controllo al suo chiamante, il record di attivazione per tale chiamata di funzione viene rimosso dallo stack e le sue variabili locali non sono più accessibili nel programma. Se una variabile locale contiene l'unico riferimento ad un oggetto nel programma, quando il record di attivazione che la contiene viene rimosso dallo stack, l'oggetto in questione non può più essere acceduto dal programma.

Naturalmente il quantitativo di memoria in un computer è finito; quindi soltanto una porzione limitata di essa può essere utilizzata per memorizzare i record di attivazione sullo stack di esecuzione del programma. Nel caso in cui si verifichino più chiamate di funzione di quanti record di attivazione possano essere memorizzati sullo stack di esecuzione del programma, si verifica un errore noto come *overflow dello stack*.

5.8 I file di intestazione

Ogni libreria standard ha un corrispondente *file di intestazione* che contiene i prototipi per tutte le funzioni incluse in quella libreria, oltre alle definizioni dei vari tipi di dato e delle costanti necessari a quelle funzioni. La Figura 5.6 elenca alfabeticamente i file di intestazione della libreria standard che potranno essere inclusi nei programmi. Il termine "macro", usato diverse volte nella Figura 5.6, sarà trattato in dettaglio nel Capitolo 13, Il preprocessore del C.

Il programmatore potrà creare dei file di intestazione personalizzati. Anche i file di intestazione definiti dal programmatore dovranno terminare in .h e potranno essere inclusi utilizzando la direttiva del preprocessore #include. Per esempio, se il prototipo per la nostra funzione square si trovasse nel file di intestazione square.h, potremmo includere quest'ultimo nel nostro programma utilizzando la seguente direttiva all'inizio del programma stesso:

```
#include "square.h"
```

La Sezione 13.2 fornirà ulteriori informazioni sull'inclusione dei file di intestazione.

5.9 Invocare le funzioni: chiamata per valore e per riferimento

In molti linguaggi di programmazione esistono due modi per invocare le funzioni: la *chiamata per valore* e la *chiamata per riferimento*. Quando gli argomenti saranno passati in una chiamata per valore, sarà preparata una *copia* dei loro valori e questa sarà passata alla funzione chiamata. Le modifiche effettuate alla copia non interesseranno il valore originale della variabile definita nel chiamante. Quando un argomento sarà passato in una chiamata per riferimento, il chiamante consentirà effettivamente alla funzione chiamata di modificare il valore originale della variabile.

Le chiamate per valore dovrebbero essere utilizzate ogniqualvolta la funzione chiamata non abbia la necessità di modificare il valore della variabile originale definita dal chiamante. Ciò preverrà gli *effetti collaterali* accidentali che intralciano pesantemente lo sviluppo di siste-

mi software corretti e affidabili. Le chiamate per riferimento dovrebbero essere utilizzate solo con funzioni affidabili che abbiano bisogno di modificare la variabile originale.

In C, tutte le chiamate di funzione sono per valore. È però possibile simulare la chiamata per riferimento, come vedremo nel Capitolo 7, utilizzando gli operatori di indirizzo e quelli di deriferimento. Nel Capitolo 6 vedremo che i vettori sono passati automaticamente con una chiamata per riferimento simulata. Dovremo aspettare fino al Capitolo 7 per una piena comprensione di questo complesso argomento. Per ora, ci concentreremo sulla chiamata per valore.

5.10 Generazione di numeri casuali

Ci prenderemo ora un breve e, si spera, divertente diversivo con una popolare applicazione della programmazione, vale a dire la simulazione e il gioco. In questa e nella prossima sezione, svilupperemo un programma di gioco ben strutturato che includerà molte funzioni. Il programma utilizzerà molte delle strutture di controllo che abbiamo studiato.

C'è qualcosa nell'aria di un casinò che rinvigorisce ogni tipo di persona: dai giocatori incalliti che attorniano i lussuosi tavoli in mogano e feltro per il gioco dei dadi a quelli che infilano monetine nelle macchinette mangiasoldi. È l'*elemento della casualità*, la possibilità che questa possa convertire una manciata di monete in una montagna di ricchezze. L'elemento della casualità può essere introdotto nelle applicazioni per computer, utilizzando la funzione `rand` della libreria standard del C.

Considerate la seguente istruzione:

```
i = rand();
```

La funzione `rand` genera un intero compreso tra 0 e `RAND_MAX` (una costante simbolica definita nel file di intestazione `<stdlib.h>`). Lo standard dell'ANSI stabilisce che il valore di `RAND_MAX` debba essere almeno 32767, che è anche il valore massimo per un intero di due byte (ovverosia 16 bit). I programmi di questa sezione sono stati collaudati su un sistema C con un valore massimo per `RAND_MAX` di 32767. Qualora `rand` generasse davvero degli interi a caso, ogni numero tra 0 e `RAND_MAX` avrebbe la stessa *possibilità* (o *probabilità*) di essere scelto ogni volta che `rand` fosse richiamata.

L'intervallo dei valori prodotti direttamente da `rand` è spesso differente da quello che sarà necessario in una specifica applicazione. Per esempio, un programma che simuli il lancio di una monetina potrebbe richiedere soltanto 0 per "testa" e 1 per "croce". Un programma che simuli il lancio di un dado a sei facce richiederebbe degli interi compresi nell'intervallo da 1 a 6.

Per mostrare l'utilizzo di `rand`, svilupperemo un programma che simulerà 20 lanci di un dado a sei facce e visualizzeremo il valore ottenuto da ogni lancio. Il prototipo per la funzione `rand` può essere ritrovato in `<stdlib.h>`. Utilizzeremo l'operatore resto (`%`) in congiunzione con `rand` nel modo seguente:

```
rand() % 6
```

per generare degli interi compresi nell'intervallo da 0 a 5. Questa operazione si chiama *riduzione in scala*, mentre il numero 6 è detto *fattore di scala*. In seguito trasleremo l'intervallo dei numeri generati, aggiungendo 1 al nostro risultato precedente. L'output nella Figura 5.7 conferma che i risultati sono compresi nell'intervallo da 1 a 6.

Per mostrare che questi numeri si presenteranno approssimativamente con la stessa probabilità, simuleremo 6000 lanci di un dado con il programma della Figura 5.8. Ogni intero da 1 a 6 dovrebbe presentarsi approssimativamente 1000 volte.

```
1   /* Fig. 5.7: fig05_07.c
2      Gli interi, ridotti in scala e traslati, generati da 1 + rand() % 6 */
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   /* l'esecuzione del programma inizia dalla funzione main */
7   int main()
8   {
9      int i; /* contatore */
10
11     /* itera 20 volte */
12     for ( i = 1; i <= 20; i++ ) {
13
14        /* genera un numero casuale da 1 a 6 e lo visualizza */
15        printf( "%10d", 1 + ( rand() % 6 ) );
16
17        /* se il contatore è divisibile per 5, inizia una nuova
             linea nella visualizzazione */
18        if ( i % 5 == 0 ) {
19           printf( "\n" );
20        } /* fine del comando if */
21
22     } /* fine del comando for */
23
24     return 0; /* indica che il programma è terminato con successo */
25
26  } /* fine della funzione main */
```

```
     6         6         5         5         6
     5         1         1         5         3
     6         6         2         4         2
     6         2         3         4         1
```

Figura 5.7 Gli interi, ridotti in scala e traslati, generati da 1 + rand() % 6

```
1   /* Fig. 5.8: fig05_08.c
2      Lancia un dado a sei facce 6000 volte */
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   /* l'esecuzione del programma inizia dalla funzione main */
7   int main()
8   {
9      int frequency1 = 0; /* contatore relativo al punteggio 1 */
10     int frequency2 = 0; /* contatore relativo al punteggio 2 */
```

```
11      int frequency3 = 0; /* contatore relativo al punteggio 3 */
12      int frequency4 = 0; /* contatore relativo al punteggio 4 */
13      int frequency5 = 0; /* contatore relativo al punteggio 5 */
14      int frequency6 = 0; /* contatore relativo al punteggio 6 */
15
16      int roll; /* contatore del numero di lanci, il valore varia
                     da 1 a 6000 */
17      int face; /* rappresenta l'esito di un lancio del dado, il valore
                     varia da 1 a 6 */
18
19      /* itera 6000 volte e riassume I risultati */
20      for ( roll = 1; roll <= 6000; roll++ ) {
21         face = 1 + rand() % 6; /* numero casuale da 1 a 6 */
22
23         /* determina il valore della faccia del dado e incrementa
               il contatore corrispondente */
24         switch ( face ) {
25
26            case 1: /* è uscito il punteggio 1 */
27               ++frequency1;
28               break;
29
30            case 2: /* è uscito il punteggio 2 */
31               ++frequency2;
32               break;
33
34            case 3: /* è uscito il punteggio 3 */
35               ++frequency3;
36               break;
37
38            case 4: /* è uscito il punteggio 4 */
39               ++frequency4;
40               break;
41
42            case 5: /* è uscito il punteggio 5 */
43               ++frequency5;
44               break;                                          45
46            case 6: /* è uscito il punteggio 6 */
47               ++frequency6;
48               break; /* opzionale */
49         } /* fine del comando switch */
50
51      } /* fine del comando for */
52
53      /* visualizza i risultati in formato tabulare */
54      printf( "%s%13s\n", "Face", "Frequency" );
55      printf( "   1%13d\n", frequency1 );
56      printf( "   2%13d\n", frequency2 );
57      printf( "   3%13d\n", frequency3 );
```

Figura 5.8 Lanciare 6000 volte un dado a sei facce (continua)

```
58      printf( "   4%13d\n", frequency4 );
59      printf( "   5%13d\n", frequency5 );
60      printf( "   6%13d\n", frequency6 );
61
62      return 0; /* indica che il programma è terminato con successo */
63
64   } /* fine della funzione main */
```

```
Face    Frequency
   1         1003
   2         1017
   3          983
   4          994
   5         1004
   6          999
```

Figura 5.8 Lanciare 6000 volte un dado a sei facce

Riducendo in scala e traslando l'intervallo, come mostra l'output del programma, abbiamo utilizzato la funzione `rand` per simulare realisticamente il lancio di un dado a sei facce. Osservate che non è stato fornito *nessun* caso di `default` nel comando `switch`. Osservate anche l'utilizzo della specifica di conversione `%s` per visualizzare le stringhe di caratteri `"Face"` e `"Frequency"` come intestazioni di colonna (riga 54). Dopo aver studiato i vettori nel Capitolo 6, mostreremo come sostituire elegantemente l'intero comando `switch` con un'istruzione di una sola riga. Eseguendo nuovamente il programma della Figura 5.7 si produrrà

```
6     6     5     5     6
5     1     1     5     3
6     6     2     4     2
6     2     3     4     1
```

Osservate che è stata visualizzata la stessa sequenza di valori. Come è possibile che siano dei numeri casuali? Ironicamente, questa riproducibilità è essenziale per dimostrare che le correzioni a un programma funzionano appropriatamente.

La funzione `rand` genera in realtà dei *numeri pseudocasuali*. Richiamando ripetutamente `rand`, si produrrebbe una sequenza di numeri che sembra essere casuale. In realtà, la sequenza si ripeterà ogni volta che il programma sarà eseguito. Una volta che il programma sarà stato completamente messo a punto, potrà essere condizionato in modo da produrre una sequenza diversa di numeri casuali a ogni sua esecuzione. Questa operazione è detta in gergo *randomizzazione* (da random, a caso) ed è eseguita con la funzione `srand` della libreria standard. La funzione `srand` riceve un argomento intero `unsigned` e *insemina* la funzione `rand`, in modo da indurla a generare una diversa sequenza di numeri casuali a ogni esecuzione del programma.

Nella Figura 5.9 è mostrato l'utilizzo della funzione `srand`. Nel programma utilizziamo il tipo di dato `unsigned`, che è un'abbreviazione per `unsigned int`. Un `int` è immagazzinato in almeno due byte di memoria e può contenere valori positivi e negativi. Anche una variabile di tipo `unsigned` è immagazzinata in almeno due byte di memoria. Un `unsigned int` di due byte può contenere soltanto valori positivi compresi nell'intervallo da 0 a 65535.

Un `unsigned int` di quattro byte può avere solo valori positivi compresi nell'intervallo da 0 a 4294967295. La funzione `srand` accetta come argomento un valore `unsigned`. La specifica di conversione `%u` è utilizzata per leggere un valore `unsigned` con la funzione `scanf`. Il prototipo di funzione per `srand` si trova in `<stdlib.h>`.

Eseguiamo diverse volte il programma e analizziamo i risultati. Osservate che si otterrà una sequenza *diversa* di numeri casuali ogni volta che il programma sarà eseguito, purché venga fornito un seme differente.

Qualora volessimo randomizzare senza la necessità di immettere un seme ogni volta potremmo utilizzare un'istruzione come:

```
srand( time( NULL ) );
```

Questa indurrebbe il computer a leggere il suo orologio interno per ottenere automaticamente un valore per il seme. La funzione `time` restituisce l'ora corrente del giorno espressa in secondi. Questo valore sarà convertito in un intero senza segno e utilizzato come seme per il generatore di numeri casuali. La funzione `time` riceverà come argomento `NULL` (normalmente `time` è in grado di fornire al programmatore una stringa che rappresenta l'ora del giorno; il `NULL` disabilita appunto questa capacità per una chiamata specifica di `time`). Il prototipo di funzione per `time` è in `<time.h>`.

```
1  /* Fig. 5.9: fig05_09.c
2     Randomizzare il programma di lancio del dado */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  /* l'esecuzione del programma inizia dalla funzione main */
7  int main()
8  {
9     int i; /* contatore */
10    unsigned seed; /* numero utilizzato come seme per il
generatore
                        di numeri casuali */
11
12    printf( "Enter seed: " );
13    scanf( "%u", &seed ); /* si noti la specifica %u per interi
                               senza segno */
14
15    srand( seed ); /* seme per il generatore di numeri casuali */
16
17    /* itera 10 volte */
18    for ( i = 1; i <= 10; i++ ) {
19
20       /* genera un numero casuale fra 1 e 6 e lo visualizza */
21       printf( "%10d", 1 + ( rand() % 6 ) );
22
23       /* se il contatore è divisibile per 5, inizia una nuova
            linea di output */
24       if ( i % 5 == 0 ) {
25          printf( "\n" );
```

Figura 5.9 Randomizzare il programma di lancio del dado (continua)

```
26          } /* fine del comando if */
27
28       } /* fine del comando for */
29
30       return 0; /* indica che il programma è terminato con successo */
31
32    } /* fine della funzione main */
```

```
Enter seed: 67
6       1       4       6       2
1       6       1       6       4
```

```
Enter seed: 867
2       4       6       1       6
1       1       3       6       2
```

```
Enter seed: 67
6       1       4       6       2
1       6       1       6       4
```

Figura 5.9 Randomizzare il programma di lancio del dado

I valori prodotti direttamente da `rand` sono sempre compresi nell'intervallo:

```
0 = rand() = RAND_MAX
```

In precedenza abbiamo mostrato come scrivere una singola istruzione C per simulare il lancio di un dado a sei facce:

```
face = 1 + rand() % 6;
```

Questa istruzione assegnerà alla variabile `face` sempre un valore intero (scelto a caso) compreso nell'intervallo $1 \leq$ `face` ≤ 6. Osservate che l'ampiezza di questo intervallo, in altre parole la quantità di interi consecutivi compresi nello stesso, sarà 6 e che il numero di partenza sarà 1. Facendo riferimento all'istruzione precedente, noteremo che l'ampiezza dell'intervallo sarà determinata dal valore utilizzato per ridurre in scala `rand` con l'operatore resto (ovverosia 6), e che il numero di partenza dell'intervallo sarà uguale a quello che sarà stato aggiunto a `rand % 6` (ovverosia 1). Possiamo generalizzare questo risultato come segue:

```
n = a + rand() % b;
```

dove `a` sarà il *valore di traslazione* (che sarà uguale al primo numero dell'intervallo di interi consecutivi desiderato), mentre `b` sarà il fattore di scala (che sarà uguale alla ampiezza dell'intervallo di interi consecutivi desiderato). Negli esercizi, vedremo che sarà possibile scegliere degli interi a caso provenienti da insiemi di valori diversi dagli intervalli di interi consecutivi.

Errore tipico 5.11

Utilizzare `srand` *invece di* `rand` *per generare dei numeri casuali.*

5.11 Esempio: un gioco d'azzardo

Uno dei giochi d'azzardo più popolari è quello dei dadi conosciuto come "craps" (tiro sfortunato), che è giocato nei casinò e nei vicoli di tutto il mondo. Le regole del gioco sono semplici:

> *Un giocatore lancia due dadi. Ogni dado ha sei facce. Queste facce contengono 1, 2, 3, 4, 5 e 6 puntini. Dopo che i dadi si saranno fermati, si calcolerà la somma dei puntini sulle due facce rivolte verso l'alto. Nel caso la somma fosse 7 o 11 al primo tiro, il giocatore avrebbe vinto. Nel caso che la somma fosse 2, 3 o 12 al primo tiro (detta "craps"), il giocatore avrebbe perso (ovverosia avrebbe vinto il "banco"). Nel caso la somma fosse 4, 5, 6, 8, 9 o 10 al primo tiro, allora quella somma diventerebbe il "punteggio" del giocatore. Per vincere, si deve continuare a lanciare i dadi fino a "ottenere il vostro punteggio". Il giocatore perderebbe qualora realizzasse un 7 prima di ottenere il proprio punteggio.*

Il programma nella Figura 5.10 simulerà il suddetto gioco di dadi. La Figura 5.11 mostra diverse esecuzioni di esempio.

Osservate che, stando alle regole del gioco, il giocatore dovrà lanciare due dadi con il primo lancio e dovrà fare altrettanto più tardi con quelli successivi. Definiremo una funzione `rollDice` per lanciare i dadi e calcolare e visualizzare la loro somma. La funzione `rollDice` sarà definita una sola volta, ma sarà invocata in due punti del programma (righe 23 e 51). È interessante notare che `rollDice` non riceverà argomenti e che perciò abbiamo indicato `void` nella sua lista di parametri (riga 80). La funzione `rollDice` restituirà la somma dei due dadi, perciò nell'intestazione della funzione sarà indicato un tipo di ritorno `int`.

Il gioco è abbastanza complicato. Il giocatore potrà vincere o perdere con il primo lancio, oppure potrà vincere o perdere con ognuno di quelli successivi. La variabile `gameStatus`, definita come variabile di un nuovo tipo `enum Status`, memorizza lo stato corrente. La riga 8 crea un tipo, chiamato *enumerazione*, definito dal programmatore. Un'enumerazione, introdotta dalla parola chiave `enum`, è un insieme di costanti intere rappresentate da identificatori. Le *costanti di enumerazione* sono talvolta chiamate costanti simboliche, cioè, costanti rappresentate sotto forma di simboli. In un tipo `enum` i valori cominciano da `0` e sono incrementati di `1`. In riga 8 la costante `CONTINUE` corrisponde al valore `0`, `WON` corrisponde al valore `1` e `LOST` corrisponde al valore `2`. Risulta anche possibile assegnare un valore intero a ogni identificatore in un tipo `enum` (si veda il Capitolo 13). In un'enumerazione gli identificatori devono essere unici, ma i valori possono essere duplicati.

Quando il gioco sarà stato vinto, al primo lancio o a quelli successivi, `gameStatus` sarà impostata a `WON`. Quando il gioco sarà stato perso, al primo lancio o a quelli successivi, `gameStatus` sarà impostata a `LOST`. In tutti gli altri casi `gameStatus` sarà impostata a `CONTINUE` e il gioco continuerà.

```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* costanti di enumerazione che rappresentano lo stato del gioco */
8  enum Status { CONTINUE, WON, LOST };
```

Figura 5.10 Programma per simulare il gioco dei dadi craps (continua)

```
9
10  int rollDice( void );
11
12  /* l'esecuzione del programma inizia dalla funzione main */
13  int main()
14  {
15     int sum; /* somma del punteggio del lancio dei dadi */
16     int myPoint; /* punti guadagnati */
17
18     enum Status gameStatus; /* può contenere CONTINUE, WON o LOST */
19
20     /* "randomizza" il generatore di numeri casuali usando l'ora
          corrente */
21     srand( time( NULL ) );
22
23     sum = rollDice();            /* primo lancio del dado */
24
25     /* determina lo stato del gioco in base al punteggio dei dadi */
26     switch( sum ) {
27
28        /* vince al primo lancio */
29        case 7:
30        case 11:
31           gameStatus = WON;
32           break;
33
34        /* perde al primo lancio */
35        case 2:
36        case 3:
37        case 12:
38           gameStatus = LOST;
39           break;
40
41        /* memorizza il punteggio */
42        default:
43           gameStatus = CONTINUE;
44           myPoint = sum;
45           printf( "Point is %d\n", myPoint );
46           break; /* opzionale */
47     } /* fine del comando switch */
48
49     /* finché il gioco non è completato */
50     while ( gameStatus == CONTINUE ) {
51        sum = rollDice(); /* continua a lanciare */
52
53        /* determina lo stato del gioco */
54       if ( sum == myPoint ) {      /* vince ottenendo il punteggio */
55           gameStatus = WON; /* il gioco è finito, il giocatore
                                  ha vinto */
```

Figura 5.10 Programma per simulare il gioco dei dadi craps (continua)

```
56          } /* fine del ramo if */
57          else {
58
59             if ( sum == 7 ) {          /* perde ottenendo un 7 */
60                gameStatus = LOST; /* il gioco è finito, il giocatore
                                        ha perso */
61             } /* fine del comando if */
62
63          } /* fine del ramo else */
64
65       } /* fine del comando while */
66
67       /* visualizza un messaggio di vittoria o di sconfitta */
68       if ( gameStatus == WON ) { /* il giocatore ha vinto? */
69          printf( "Player wins\n" );
70       } /* fine del ramo if */
71       else { /* il giocatore ha perso */
72          printf( "Player loses\n" );
73       } /* fine del ramo else */
74
75       return 0; /* indica che il programma è terminato con successo */
76
77    } /* fine della funzione main */
78
79    /* lancia i dadi, calcola la somma e visualizza i risultati */
80    int rollDice( void )
81    {
82       int die1; /* punteggio del primo dado */
83       int die2; /* punteggio del secondo dado */
84       int workSum; /* somma dei punteggi dei dadi */
85
86       die1 = 1 + ( rand() % 6 ); /* genera il punteggio casuale di die1 */
87       die2 = 1 + ( rand() % 6 ); /* genera il punteggio casuale di die2 */
88       workSum = die1 + die2;
89
90       /* visualizza i risultati di questo lancio */
91       printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
92
93       return workSum; /* restituisce la somma del punteggio dei dadi */
94
95    } /* fine della funzione rollDice */
```

Figura 5.10 Programma per simulare il gioco dei dadi craps

```
Player rolled 6 + 5 = 11
Player wins
```

Figura 5.11 Esecuzioni di esempio per il gioco dei dadi craps (continua)

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

```
Player rolled 1 + 1 = 2
Player loses
```

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

Figura 5.11 Esecuzioni di esempio per il gioco dei dadi craps

Dopo il primo lancio, qualora il gioco fosse finito, il comando `while` (riga 50) sarebbe ignorato perché `gameStatus` non sarebbe uguale a `CONTINUE`. Il programma procederà con il comando `if...else` alla riga 68 che visualizzerà "`Player wins`" se `gameStatus` varrà `WON` e "`Player loses`" altrimenti.

Dopo il primo lancio, qualora il gioco non fosse ancora finito, `sum` sarebbe memorizzata in `myPoint`. L'esecuzione procederà con il comando `while` (riga 50) perché `gameStatus` varrà ancora `CONTINUE`. Per produrre una nuova `sum`, `rollDice` sarà invocata a ogni iterazione della struttura `while`. Qualora `sum` corrispondesse a `myPoint`, `gameStatus` sarebbe impostata a `WON` per indicare la vittoria del giocatore, il controllo dell'istruzione `while` fallirebbe, il comando `if...else` (riga 68) visualizzerebbe "`Player wins`" e l'esecuzione terminerebbe. Qualora `sum` fosse uguale a `7` (riga 59), `gameStatus` sarebbe impostata a `LOST` per indicare la sconfitta del giocatore, il controllo del `while` fallirebbe, l'istruzione `if...else` (riga 68) visualizzerebbe "`Player loses`" e l'esecuzione terminerebbe.

Osservate l'interessante struttura di controllo del programma. Abbiamo utilizzato due funzioni, `main` e `rollDice`, e i comandi `switch`, `while`, `if...else` nidificati e degli `if` nidificati. Negli esercizi investigheremo su varie interessanti caratteristiche del gioco dei dadi.

5.12 Le classi di memoria

Nei Capitoli dal 2 al 4, abbiamo utilizzato degli identificatori per i nomi di variabile. Gli attributi delle variabili includono il nome, il tipo e il valore. In questo capitolo, utilizzeremo gli identificatori anche come nomi per le funzioni definite dall'utente. In realtà, ogni identificatore in un programma ha anche altri attributi che includono: la *classe di memoria*, la *permanenza (o durata) in memoria*, la *visibilità* e il *collegamento*.

Il C fornisce quattro classi di memoria indicate dalle *specifiche di classe di memoria:* `auto`, `register`, `extern` e `static`. La *permanenza in memoria* di un identificatore è il periodo durante il quale quell'identificatore esiste in memoria. Alcuni identificatori esistono per un tempo limitato, altri sono creati e distrutti ripetutamente, mentre altri ancora esistono durante l'intera esecuzione del programma. La *visibilità* di un identificatore è il punto del pro-

gramma in cui si può far riferimento all'identificatore. Si potrà far riferimento ad alcuni identificatori in tutto il programma, mentre ad altri si potrà farlo solo da determinate porzioni del programma. Per un programma formato da diversi file sorgente, un argomento che tratteremo nel Capitolo 14, il *collegamento* di un identificatore determina se un identificatore sia noto solo in quello corrente o in qualsiasi altro file sorgente che abbia le dichiarazioni appropriate. Questa sezione discuterà delle quattro classi di memoria e della permanenza all'interno della stessa. La Sezione 5.13 discuterà invece della visibilità degli identificatori. Il Capitolo 14, "Argomenti avanzati", affronterà i temi del collegamento dell'identificatore e della programmazione con diversi file sorgente.

Le quattro specifiche di classe di memoria possono essere suddivise in due tipi di permanenza: la *permanenza automatica in memoria* e la *permanenza statica in memoria*. Le parole chiave `auto` e `register` sono utilizzate per dichiarare variabili con permanenza automatica in memoria. Le variabili con permanenza automatica nella memoria saranno create ogniqualvolta si entrerà nel blocco in cui sono state dichiarate, esisteranno finché quello resterà attivo e saranno distrutte quando si uscirà dallo stesso.

Solo le variabili possono avere una permanenza automatica nella memoria. Le variabili locali di una funzione (quelle dichiarate nella lista dei parametri o nel corpo della funzione) hanno normalmente una permanenza automatica nella memoria. La parola chiave `auto` dichiarerà esplicitamente delle variabili con permanenza automatica nella memoria. Per esempio, la seguente dichiarazione indica che `double x` e `y` saranno delle variabili locali automatiche e che esisteranno solo nel corpo della funzione in cui comparirà la dichiarazione:

```
auto double x, y;
```

Le variabili locali hanno per default una permanenza automatica nella memoria, perciò la parola chiave `auto` sarà utilizzata raramente. Per il resto del testo, ci riferiremo alle variabili con una permanenza automatica nella memoria chiamandole semplicemente *variabili automatiche*.

Obiettivo efficienza 5.1

La memorizzazione automatica è un mezzo per risparmiare la memoria, poiché le variabili automatiche esisteranno solo fintanto che saranno necessarie. Esse saranno create quando si entrerà nella funzione in cui sono state dichiarate e saranno distrutte quando si uscirà dalla stessa.

Ingegneria del software 5.10

La memorizzazione automatica è un esempio del principio del minimo privilegio *(a ogni istante un utente o un processo deve avere tutti e solo i diritti necessari per proseguire la computazione): permettere l'accesso ai dati soltanto quando questi ultimi sono assolutamente necessari. Per quale motivo le variabili dovrebbero essere conservate in memoria ed essere accessibili quando, di fatto, non sono necessarie?*

I dati per i calcoli e le altre elaborazioni, nella versione in linguaggio macchina di un programma, sono caricati normalmente nei registri.

Obiettivo efficienza 5.2

La specifica di classe di memoria `register` *potrà essere inserita prima della dichiarazione di una variabile automatica, per suggerire al compilatore di conservare la variabile in uno*

dei registri hardware ad alta velocità del computer. Qualora le variabili utilizzate intensamente, come i contatori o i totali, potessero essere mantenute nei registri hardware, potrebbe essere eliminato il costo ripetuto dovuto al caricamento delle variabili dalla memoria nei registri e all'immagazzinamento dei risultati di nuovo nella memoria.

Il compilatore può ignorare le dichiarazioni `register`. Per esempio, potrebbe non esserci un numero sufficiente di registri disponibili all'utilizzo del compilatore. La dichiarazione seguente suggerisce che la variabile intera `counter` sia sistemata in uno dei registri del computer e inizializzata a 1:

```
register int counter = 1;
```

La parola chiave `register` può essere utilizzata solo con le variabili con permanenza automatica nella memoria.

Obiettivo efficienza 5.3

Spesso le dichiarazioni `register` *non sono necessarie. Le procedure di ottimizzazione dei compilatori di oggigiorno sono in grado di riconoscere le variabili utilizzate più frequentemente e possono decidere di sistemarle nei registri, senza che si renda necessaria una dichiarazione* `register` *da parte del programmatore.*

Le parole chiave `extern` e `static` sono utilizzate per dichiarare degli identificatori per variabili e funzioni con una permanenza statica nella memoria. Gli identificatori con permanenza statica nella memoria esistono dal momento in cui il programma inizia la sua esecuzione. Per le variabili, la memoria sarà allocata e inizializzata solo una volta, quando il programma inizierà la propria esecuzione. Per le funzioni, il nome della funzione esisterà sin dal momento in cui il programma inizierà la propria esecuzione. Tuttavia, sebbene le variabili e i nomi delle funzioni esistano sin dall'inizio dell'esecuzione del programma, ciò non significa che questi identificatori possano essere utilizzati in tutto il programma. La permanenza nella memoria e la visibilità (in quali punti del programma può essere utilizzato un nome) sono cose distinte, come vedremo nella Sezione 5.13.

Esistono due tipi di identificatori con permanenza statica nella memoria: gli identificatori esterni (come le variabili e i nomi di funzione globali) e le variabili locali dichiarate con la specifica di classe di memoria `static`. Le variabili e i nomi di funzioni globali sono definiti per default con la classe di memoria `extern`. Le variabili globali saranno create, inserendo le loro dichiarazioni all'esterno di ogni definizione di funzione e conserveranno i propri valori durante tutta l'esecuzione del programma. Le variabili e le funzioni globali potranno essere oggetto di riferimenti da parte di qualsiasi funzione, purché questa sia successiva alle loro dichiarazioni o definizioni all'interno del file. Questa è una delle ragioni per l'utilizzo dei prototipi di funzione: nel momento in cui includeremo `stdio.h` in un programma che richiami `printf`, il prototipo della funzione sarà sistemato all'inizio del nostro file in modo da far conoscere al resto dello stesso il nome `printf`.

Ingegneria del software 5.11

Una variabile dichiarata globale, invece che locale, consentirà il verificarsi di effetti collaterali indesiderati, qualora una funzione che non avesse bisogno di accedervi la modificasse accidentalmente o maliziosamente. In generale, l'utilizzo delle variabili glo-

bali dovrebbe essere evitato, tranne che in certe situazioni in cui sia richiesta un'efficienza particolare (come sarà discusso nel Capitolo 14).

Ingegneria del software 5.12

Le variabili utilizzate esclusivamente all'interno di una particolare funzione dovrebbero essere dichiarate locali a quella, invece che esterne.

Anche le variabili locali dichiarate con la parola chiave `static` saranno note solo alla funzione in cui sono definite ma, a differenza di quelle automatiche, le variabili locali statiche conserveranno il loro valore anche quando si sarà usciti dalla funzione. Alla successiva invocazione della funzione, la variabile statica locale conterrà ancora il valore dell'ultima volta che si è usciti dalla funzione. L'istruzione successiva dichiarerà la variabile locale `count` di tipo `static` e la inizializzerà a 1.

```
static int count = 1;
```

Tutte le variabili numeriche con permanenza statica nella memoria saranno inizializzate a zero, qualora non siano state inizializzate esplicitamente dal programmatore.

Errore tipico 5.13

Utilizzare diverse specifiche di classe di memoria per un identificatore. A un identificatore può essere applicata solo una specifica di classe di memoria.

Le parole riservate `extern` e `static` avranno un significato speciale quando saranno applicate esplicitamente agli identificatori esterni. Nel Capitolo 14, "Argomenti avanzati", discuteremo dell'utilizzo esplicito di `extern` e di `static` con gli identificatori esterni e i programmi formati da molti file sorgente.

5.13 Le regole di visibilità

La *visibilità* di un identificatore è la porzione del programma in cui quello potrà essere oggetto di riferimenti. Per esempio, quando in un blocco dichiareremo una variabile locale, questa potrà essere oggetto di riferimenti solo all'interno di quel blocco o di quelli che vi si annidano. I quattro tipi di visibilità possibili per un identificatore sono: *visibilità nella funzione*, *visibilità nel file*, *visibilità nel blocco* e *visibilità nel prototipo di funzione*.

Le etichette (un identificatore seguito da un due punti, come `start:`) sono gli unici identificatori che hanno una *visibilità nella funzione*. Le etichette potranno essere utilizzate in qualsiasi parte della funzione in cui compaiono, ma non potranno essere oggetto di riferimenti esterni al suo corpo. Le etichette sono utilizzate nelle strutture `switch` (per esempio, quelle dei `case`) e nelle istruzioni `goto` (consultate il Capitolo 14, "Argomenti avanzati"). Le etichette sono dei dettagli di implementazione che le funzioni si nascondono l'una con l'altra. Questo occultamento, detto più formalmente *incapsulamento delle informazioni,* è uno dei principi fondamentali di una buona ingegneria del software.

Un identificatore dichiarato all'esterno di qualsiasi funzione ha una *visibilità nel file*. Un tale identificatore sarà "noto" (cioè, accessibile) in tutte le funzioni incluse fra il punto in cui quello è stato dichiarato e la fine del file. Le variabili globali, le definizioni e i prototipi di funzione inseriti all'esterno di una funzione hanno tutti visibilità nel file.

Gli identificatori dichiarati all'interno di un blocco hanno una *visibilità nel blocco*. La visibilità nel blocco termina con la parentesi graffa (}) di chiusura dello stesso. Avranno quindi visibilità nel blocco le variabili locali che sono state dichiarate all'inizio di una funzione, come pure i suoi parametri, che quella considera alla stregua di variabili locali. Qualsiasi blocco potrà contenere delle dichiarazioni di variabili. Qualora i blocchi fossero nidificati e un identificatore di quello più esterno avesse lo stesso nome di un identificatore di quello più interno, l'identificatore di quello più esterno sarebbe "nascosto" fino alla fine del blocco più interno. Ciò significa che, durante l'esecuzione del blocco più interno, questo vedrà il valore del suo identificatore locale e non quello dell'identificatore omonimo dichiarato nel blocco che lo racchiude. Anche le variabili locali dichiarate `static` hanno una visibilità nel blocco, sebbene esistano sin dal momento in cui inizia l'esecuzione del programma. Ne consegue che la permanenza in memoria non influisce sulla visibilità di un identificatore.

Gli unici identificatori con una *visibilità nel prototipo di funzione* sono quelli utilizzati nella lista dei parametri inclusa in un prototipo di funzione. Come menzionato in precedenza, i prototipi di funzione non richiedono dei nomi nella lista dei parametri: solo i tipi sono obbligatori. Il compilatore ignorerà ogni nome incluso nella lista dei parametri di un prototipo di funzione. Gli identificatori utilizzati in un prototipo di funzione potranno essere riutilizzati altrove nel programma, senza generare delle ambiguità.

Errore tipico 5.14

Usare accidentalmente, in un blocco più interno, lo stesso nome di un identificatore dichiarato in un blocco più esterno qualora il programmatore volesse, di fatto, che l'identificatore del blocco più esterno rimanesse attivo per tutta la permanenza di quello più interno.

Collaudo e messa a punto 5.2

Evitate nomi di variabile che nascondano quelli con visibilità più esterne. Ciò potrà essere ottenuto evitando semplicemente di utilizzare in un programma degli identificatori duplicati.

Il programma della Figura 5.12 illustra i problemi di visibilità relativi alle variabili globali, alle automatiche locali e a quelle locali `static`. È stata dichiarata e inizializzata a 1 una variabile globale `x` (riga 9). Questa sarà nascosta in ogni blocco (o funzione) in cui ne sarà stata dichiarata una, locale, chiamata `x`. Nella funzione `main`, è stata dichiarata e inizializzata a 5 una variabile locale `x` (riga 14). Questa sarà quindi visualizzata per dimostrare che all'interno di `main` la variabile globale `x` è nascosta. È stato quindi definito un nuovo blocco all'interno di `main` con un'altra variabile locale `x` inizializzata a 7 (riga 19). Questa variabile sarà visualizzata per dimostrare che nasconde la `x` dichiarata nel blocco più esterno di `main`. La variabile `x` con il valore 7 sarà distrutta automaticamente all'uscita dal blocco, mentre la variabile locale `x` dichiarata in quello più esterno di `main` sarà visualizzata nuovamente per dimostrare che non è più nascosta. Il programma definisce tre funzioni che non richiedono argomenti, né restituiscono dei risultati. La funzione `useLocal` dichiarerà una variabile automatica `x` e la inizializzerà a 25 (riga 42). Quando `useLocal` sarà stata invocata la variabile sarà visualizzata, incrementata e visualizzata nuovamente, prima dell'uscita dalla funzione. Ogni volta che questa funzione sarà stata invocata la variabile automatica `x` sarà inizializzata nuovamente a 25. La funzione `useStaticLocal` dichiarerà una variabile `static x` e la inizializzerà a 50 (riga 55). Le variabili locali dichiarate `static` conservano i loro valori anche quando sono al di fuori della propria visibilità. Quando `useStaticLocal` sarà stata invocata

x sarà visualizzata, incrementata e visualizzata nuovamente prima dell'uscita dalla funzione. Nell'invocazione successiva di questa funzione la variabile locale `static x` conterrà ancora il valore 51. La funzione `useGlobal` non dichiarerà alcuna variabile. Di conseguenza, quando farà riferimento a x sarà utilizzata la variabile globale (riga 9). Quando `useGlobal` sarà stata invocata la variabile globale sarà visualizzata, moltiplicata per 10 e stampata nuovamente prima dell'uscita dalla funzione. La prossima volta che la funzione `useGlobal` sarà stata invocata la variabile globale conterrà ancora il suo valore modificato: 10. Infine, il programma visualizzerà ancora una volta la variabile locale x dichiarata nel `main` (riga 33), per dimostrare che nessuna delle funzioni chiamate ne avrà modificato il valore, poiché tutte loro hanno fatto riferimento a variabili con altre visibilità.

```
1  /* Fig. 5.12: fig05_12.c
2     Un esempio di visibilità */
3  #include <stdio.h>
4
5  void useLocal( void );   /* prototipo di funzione */
6  void useStaticLocal( void );   /* prototipo di funzione */
7  void useGlobal( void );   /* prototipo di funzione */
8
9  int x = 1;      /* variabile globale */
10
11 /* l'esecuzione del programma inizia dalla funzione main */
12 int main()
13 {
14    int x = 5;   /* variabile locale a main */
15
16    printf( "local x in outer scope of main is %d\n", x );
17
18    {            /* incomincia una nuova visibilità */
19       int x = 7; /* variabile locale per la nuova visibilità */
20
21       printf( "local x in inner scope of main is %d\n", x );
22    }            /* fine della nuova visibilità */
23
24    printf( "local x in outer scope of main is %d\n", x );
25
26    useLocal();  /* useLocal ha una variabile x locale automatica */
27    useStaticLocal();  /* useStaticLocal ha una variabile x locale
                            statica */
28    useGlobal();  /* useGlobal usa la variabile x globale */
29    useLocal();   /* useLocal inizializza la variabile x locale
                       automatica */
30    useStaticLocal();  /* la variabile x locale statica conserva
                            il suo valore precedente */
31    useGlobal();  /* anche la variabile x globale conserva il suo
                       valore precedente */
32
33    printf( "\nlocal x in main is %d\n", x );
```

Figura 5.12 Un esempio di visibilità (continua)

```
34
35    return 0; /* indica che il programma è terminato con successo */
36
37 } /* fine della funzione main */
38
39 /* useLocal inizializza nuovamente la variabile locale x a ogni
      chiamata */
40 void useLocal( void )
41 {
42    int x = 25;  /* inizializzata ogni volta che useLocal è chiamata */
43
44    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
45    x++;
46    printf( "local x in useLocal is %d before exiting useLocal\n", x );
47 } /* fine della funzione useLocal */
48
49 /* useStaticLocal inizializza la variabile locale statica x
      soltanto la prima volta
50    che la funzione viene chiamata; il valore di x viene salvato
51    tra una chiamata e l'altra a questa funzione */
52 void useStaticLocal( void )
53 {
54    /* inizializzata soltanto la prima volta che useStaticLocal
         viene chiamata */
55    static int x = 50;
56
57    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
58    x++;
59    printf( "local static x is %d on exiting useStaticLocal\n", x );
60 } /* fine della funzione useStaticLocal */
61
62 /* la funzione useGlobal modifica la variabile globale x a ogni
      chiamata */
63 void useGlobal( void )
64 {
65    printf( "\nglobal x is %d on entering useGlobal\n", x );
66    x *= 10;
67    printf("global x is %d on exiting useGlobal\n", x);
68 } /* fine della funzione useGlobal */
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal
```

Figura 5.12 Un esempio di visibilità (continua)

```
global x is 1 on entering useGlobal
global x is 10 on exiting useGloblal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Figura 5.12 Un esempio di visibilità

5.14 La ricorsione

I programmi di cui abbiamo discusso sono generalmente strutturati come funzioni che si chiamano l'una con l'altra in modo disciplinato e gerarchico. Per alcuni tipi di problemi sarà invece utile avere delle funzioni che richiamino se stesse. Una *funzione ricorsiva* è una funzione che richiama se stessa direttamente o indirettamente attraverso un'altra funzione. La ricorsione è un argomento complesso discusso a lungo nei corsi superiori di informatica. In questa sezione e nella prossima, saranno presentati dei semplici esempi di ricorsione. Questo libro contiene una discussione estesa della ricorsione distribuita nei Capitoli dal 5 al 12. La Figura 5.17, nella Sezione 5.16, riassume i 31 esempi ed esercizi sulla ricorsione presenti in questo libro.In primo luogo considereremo la ricorsione dal punto di vista concettuale e quindi esamineremo diversi programmi contenenti delle funzioni ricorsive. Gli approcci ricorsivi alla soluzione dei problemi hanno un certo numero di elementi in comune. Una funzione ricorsiva sarà invocata per risolvere un problema. La funzione in realtà sa come risolvere soltanto i casi più semplici, ovverosia i cosiddetti *casi di base*. Nel caso che la funzione venga invocata su un caso di base, restituirà direttamente un risultato. Nel caso invece che la funzione venga invocata su un problema più complesso, lo suddividerà in due parti concettuali: una che la funzione sa come risolvere e un'altra che invece non sa come risolvere. Per rendere fattibile la ricorsione, la seconda parte del problema dovrà assomigliare a quello originale, ma dovrà anche essere una versione leggermente più semplice o più piccola dello stesso. Dato che questo nuovo problema assomiglierà a quello originale, la funzione lancerà (invocherà) una nuova copia di se stessa perché lavori sul problema più piccolo: questa è appunto ciò che viene definita *chiamata ricorsiva* o anche *passo ricorsivo*. Il passo ricorsivo includerà anche la parola chiave `return` perché il suo risultato sarà combinato con la porzione del problema che la funzione sa come risolvere, in modo da formare il risultato che sarà finalmente restituito al chiamante originale, probabilmente il `main`.

Il passo ricorsivo sarà eseguito fintanto che la chiamata originaria della funzione sarà ancora aperta, ovverosia non avrà ancora terminato la sua esecuzione. Il passo ricorsivo potrà generare molte altre chiamate ricorsive simili, man mano che continuerà a suddividere nelle due parti concettuali ognuno dei problemi su cui è stato invocato. Perché la ricorsione possa terminare, ogni volta che la funzione richiamerà se stessa con una versione leggermente più semplice del problema originale, questa successione di problemi sempre più piccoli dovrà

convergere alla fine a un caso di base. A quel punto, la funzione riconoscerà il caso di base, restituirà un risultato alla sua copia precedente e ne conseguirà una sequenza di restituzioni lungo tutta la catena di chiamate, finché quella della funzione originaria non restituirà finalmente al `main` il risultato finale. Tutto ciò sembra alquanto esotico se confrontato con il tipo di risoluzione dei problemi cui siamo stati abituati fino a questo punto tramite chiamate di funzioni convenzionali. Scrivere dei programmi ricorsivi, richiederà certamente una buona dose di pratica, prima che il processo appaia naturale. Come esempio di funzionamento di questi concetti, scriveremo un programma ricorsivo che eseguirà un tipico calcolo matematico.

Il fattoriale di un intero non negativo *n*, scritto *n!* (e pronunciato "*fattoriale di n*"), è il prodotto

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

con 1! uguale a 1 e 0! definito uguale a 1. Per esempio, 5! sarà il prodotto 5 * 4 * 3 * 2 * 1, che sarà uguale a 120.

Il fattoriale di un intero, `number`, maggiore o uguale a zero, potrà essere calcolato *iterativamente* (non in modo ricorsivo) utilizzando la struttura `for` come segue:

```
factorial = 1;
for ( counter = number; counter >= 1; counter— )
   factorial *= counter;
```

Si otterrà una definizione ricorsiva della funzione fattoriale, osservando la seguente relazione:

$$n! = n \cdot (n-1)!$$

Per esempio, 5! sarà chiaramente uguale a 5 * 4!, come dimostrato di seguito:

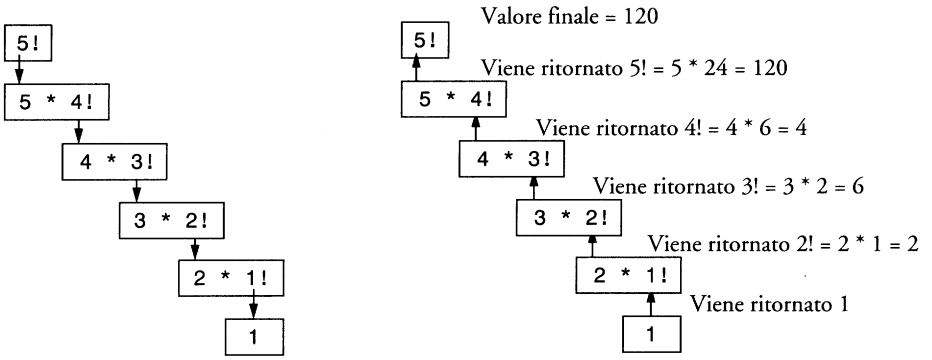
$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

La valutazione di 5! procederà nel modo mostrato nella Figura 5.13. La Figura 5.13a mostra in che modo la successione delle chiamate ricorsive procederà finché 1! sarà valutato 1 e terminerà la ricorsione. La Figura 5.13b mostra i valori restituiti da ogni chiamata ricorsiva al suo chiamante finché sarà stato calcolato e restituito il valore finale.

Il programma della Figura 5.14 utilizzerà la ricorsione, per calcolare e visualizzare il fattoriale degli interi compresi tra 0 a 10 (la scelta del tipo di dato `long` sarà spiegata tra breve). La funzione ricorsiva `factorial` controllerà in primo luogo se una delle condizioni di terminazione sia vera, ovverosia se `number` sia inferiore o uguale a 1. Nel caso che `number` sia sicuramente minore o uguale a 1, `factorial` restituirà 1, non saranno necessarie altre ricorsioni e il programma terminerà. Nel caso che `number` sia maggiore di 1, l'istruzione

```
return number * factorial( number - 1) ;
```

esprimerà il problema come il prodotto tra `number` e una chiamata ricorsiva di `factorial` che valuterà il fattoriale di `number - 1`. Osservate che `factorial( number - 1 )` sarà un problema leggermente più semplice del calcolo originale `factorial( number )`.



a) Sequenza di chiamate ricorsive.

b) Valori ritornati da ogni chiamata ricorsiva.

Figura 5.13 Valutazione ricorsiva di 5!

La funzione `factorial` (riga 23) è stata dichiarata in modo da ricevere un parametro di tipo `long` e restituire un risultato dello stesso tipo. Questa è una notazione abbreviata per `long int`. Lo standard C specifica che una variabile di tipo `long int` debba essere immagazzinata in almeno quattro byte e, di conseguenza, che possa contenere un valore grande quanto +2147483647. Come si può vedere dalla Figura 5.14, i valori del fattoriale diventano rapidamente grandi. Abbiamo scelto il tipo di dato `long` proprio perché il programma possa calcolare dei fattoriali maggiori di 7! anche su computer con interi piccoli (come quelli da 2 byte). La specifica di conversione `%ld` è stata utilizzata per visualizzare dei valori di tipo `long`. Sfortunatamente, la funzione `factorial` produrrà dei valori grandi così velocemente che anche un `long int` non ci aiuterà a visualizzare molti valori fattoriali, prima che sia superata la dimensione di una variabile di quel tipo.

In definitiva, come esamineremo negli esercizi, all'utente che desideri calcolare i fattoriali di numeri più grandi potranno essere necessari dei `float` e dei `double`. Tutto ciò rende evidente una debolezza del C (peraltro comune alla maggior parte degli altri linguaggi di programmazione), ovverosia il fatto che esso non possa essere ampliato con facilità affinché possa gestire i particolari requisiti delle varie applicazioni. Il C++ invece è un linguaggio che si può estendere e che quindi, qualora lo volessimo, ci consentirebbe di creare degli interi arbitrariamente grandi.

Errore tipico 5.15

Dimenticare di restituire un valore da una funzione ricorsiva, qualora sia necessario.

Errore tipico 5.16

Omettere un caso di base o scrivere scorrettamente il passo di ricorsione in modo tale che non converga a un caso di base provocherà delle ricorsioni infinite, finendo per esaurire la memoria. Ciò è analogo al problema del ciclo infinito di una soluzione iterativa (non ricorsiva). Una ricorsione infinita potrà essere provocata anche fornendo un input inatteso.

```
1  /* Fig. 5.14: fig05_14.c
2     Funzione fattoriale ricorsiva */
3  #include <stdio.h>
4
5  long factorial( long number ); /* prototipo di funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    int i; /* contatore */
11
12    /* itera 11 volte; a ogni iterazione calcola
13       factorial( i ) e visualizza il risultato */
14    for ( i = 0; i <= 10; i++ ) {
15       printf( "%2d! = %ld\n", i, factorial( i ) );
16    } /* fine del comando for */
17
18    return 0; /* indica che il programma è terminato con successo */
19
20 } /* fine della funzione main */
21
22 /* Definizione ricorsiva della funzione fattoriale */
23 long factorial( long number )
24 {
25    /* caso base */
26    if ( number <= 1 ) {
27       return 1;
28    } /* fine del ramo if */
29    else { /* passo ricorsivo */
30       return ( number * factorial( number - 1 ) );
31    } /* fine del ramo else */
32
33 } /* fine della funzione factorial */
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Figura 5.14 Calcolare i fattoriali con una funzione ricorsiva

5.15 Esempio di utilizzo della ricorsione: la serie di Fibonacci

La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

comincia con 0 e 1 e gode della proprietà che ogni susseguente numero di Fibonacci è la somma dei due che lo precedono.

La serie ha dei riscontri in vari fenomeni naturali e descrive in particolare una forma di spirale. Il rapporto tra i numeri consecutivi di Fibonacci converge verso il valore costante di 1,618.... Anche questo numero ha ripetuti riscontri in natura ed è stato definito *rapporto aureo* o *divina proporzione*.

Gli esseri umani hanno la tendenza a considerare la divina proporzione piacevole dal punto di vista estetico. Gli architetti progettano spesso finestre, stanze e palazzi le cui lunghezze e altezze sono in un rapporto pari alla divina proporzione. Anche le cartoline postali sono disegnate spesso con un rapporto lunghezza/larghezza pari alla divina proporzione.

La serie di Fibonacci può essere definita in modo ricorsivo come segue:

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Il programma della Figura 5.15 calcolerà in modo ricorsivo l'i-esimo numero di Fibonacci, utilizzando la funzione `fibonacci`. Osservate che i numeri di Fibonacci tendono a diventare grandi rapidamente. Per questo motivo abbiamo scelto il tipo di dato `long` per il parametro e per il valore restituito dalla funzione `fibonacci`. Nella Figura 5.15 ogni coppia di righe dell'output mostra un'esecuzione distinta del programma.

L'invocazione di `fibonacci` da parte di `main` non è una chiamata ricorsiva (riga 18), mentre lo saranno tutte le invocazioni successive di `fibonacci` (riga 35). Ogni volta che `fibonacci` sarà stata invocata, essa controllerà immediatamente il caso di base: `n` uguale a 0 o a 1. Verrebbe restituito `n` se questa condizione dovesse essere vera. In modo interessante, se `n` dovesse essere maggiore di 1, il passo della ricorsione genererebbe *due* chiamate ricorsive, ognuna delle quali sarebbe effettuata per un problema leggermente più semplice di quello della chiamata originaria a `fibonacci`. La Figura 5.16 mostra in che modo la funzione `fibonacci` valuterà la chiamata `fibonacci( 3 )`.

```
1  /* Fig. 5.15: fig05_15.c
2     Funzione ricorsiva fibonacci */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* prototipo di funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    long result; /* valore di fibonacci */
11    long number; /* numero specificato dall'utente */
```

Figura 5.15 Generare i numeri di Fibonacci in modo ricorsivo (continua)

```
12
13      /* ottiene un intero dall'utente */
14      printf( "Enter an integer: " );
15      scanf( "%ld", &number );
16
17      /* calcola il valore di fibonacci relativo al numero specificato
           dall'utente */
18      result = fibonacci( number );
19
20      /* visualizza il risultato */
21      printf( "Fibonacci( %ld ) = %ld\n", number, result );
22
23      return 0; /* indica che il programma è terminato con successo */
24
25   } /* fine della funzione main */
26
27   /* Definizione ricorsiva della funzione fibonacci */
28   long fibonacci( long n )
29   {
30      /* caso base */
31      if ( n == 0 || n == 1 ) {
32         return n;
33      } /* fine del ramo if */
34      else { /* passo ricorsivo */
35         return fibonacci( n - 1 ) + fibonacci( n - 2 );
36      } /* fine del ramo else */
37
38   } /* fine della funzione fibonacci */
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

```
Enter an integer: 4
Fibonacci( 4 ) = 3
```

```
Enter an integer: 5
Fibonacci( 5 ) = 5
```

Figura 5.15 Generare i numeri di Fibonacci in modo ricorsivo (continua)

```
Enter an integer: 6
Fibonacci( 6 ) = 8
```

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 35
Fibonacci( 35 ) = 9227465
```

Figura 5.15 Generare i numeri di Fibonacci in modo ricorsivo

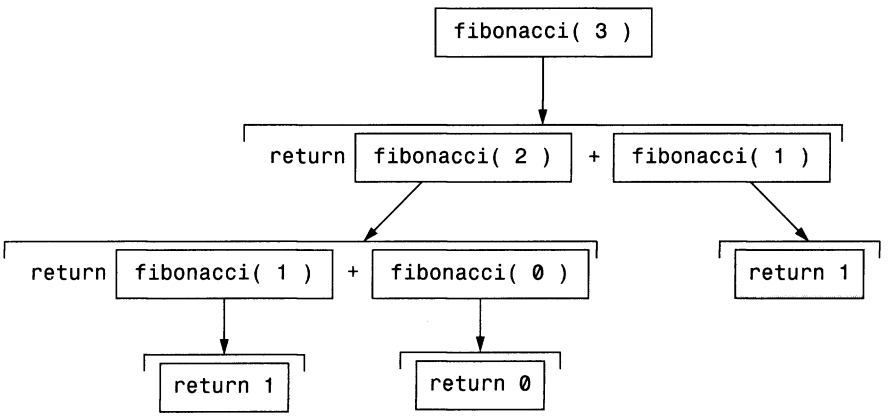


Figura 5.16 Gruppo di chiamate ricorsive alla funzione fibonacci

Questa figura solleva alcuni interessanti problemi riguardanti l'ordine in cui il compilatore C valuterà gli operandi degli operatori. Questo è un problema differente da quello relativo all'ordine in cui gli operatori saranno applicati ai propri operandi, ovverosia all'ordine dettato dalle regole di priorità degli operatori. Dalla Figura 5.16 appare evidente che nel valutare `fibonacci( 3 )` saranno effettuate due invocazioni ricorsive, ovverosia `fibonacci( 2 )` e `fibonacci( 1 )`. Ma in quale ordine saranno effettuate queste chiamate? Molti programmatori presumeranno semplicemente che gli operandi saranno valutati da sinistra a destra. Stranamente, lo standard ANSI non specifica l'ordine in cui debbano essere valutati gli operandi della maggior parte degli operatori (incluso +). Ne consegue che il programmatore non potrà presumere alcunché sull'ordine in cui saranno eseguite le suddette chiamate. Infatti `fibonacci( 2 )` potrebbe essere eseguita prima di `fibonacci( 1 )`, ma le stesse invocazioni potrebbero anche essere eseguite nell'ordine inverso, `fibonacci( 1 )` prima di `fibonacci( 2 )`. In questo e nella maggior parte dei programmi il risultato sarà lo stesso in ogni caso. In alcuni programmi, invece, la valutazione di un operando potrà produrre degli

effetti collaterali che potrebbero a loro volta influenzare il risultato finale dell'espressione. Lo standard ANSI specifica l'ordine di valutazione solo per quattro dei tanti operatori inclusi nel C, vale a dire &&, ||, l'operatore virgola (,) e ?: . I primi tre sono operatori binari e per i loro operandi sarà garantita la valutazione da sinistra a destra. [*Nota*: Le virgole utilizzate per separare gli argomenti in una chiamata di funzione non sono operatori virgola.] L'ultimo è l'unico operatore ternario del C. Il suo operando più a sinistra sarà valutato sempre per primo; qualora l'operando più a sinistra fosse diverso da zero, seguirebbe la valutazione di quello di mezzo e l'ultimo sarebbe ignorato; qualora l'operatore più a sinistra fosse valutato zero, seguirebbe invece la valutazione del terzo operando e quello di mezzo sarebbe ignorato.

Errore tipico 5.17

Scrivere dei programmi che dipendano dall'ordine di valutazione degli operandi di operatori diversi da &&, ||, ?: *e dall'operatore virgola (,) potrà provocare degli errori perché i compilatori potranno valutare gli operandi in un ordine che non sarà necessariamente quello atteso dal programmatore.*

Obiettivo portabilità 5.2

I programmi che dipendono dall'ordine di valutazione degli operandi di operatori diversi da &&, ||, ?: *e dall'operatore virgola (,) potranno funzionare in modo differente su sistemi con compilatori diversi.*

È doveroso spendere una parola di avvertimento sui programmi ricorsivi come quello che abbiamo usato qui per generare i numeri Fibonacci. Ogni livello di ricorsione della funzione fibonacci ha un effetto raddoppiante sulla quantità delle invocazioni; in altre parole, il numero delle chiamate ricorsive che saranno eseguite per calcolare l'*ennesimo* numero di Fibonacci sarà dell'ordine di 2^n. Ciò andrà rapidamente fuori controllo. Il solo calcolo del 20° numero di Fibonacci richiederà una quantità di invocazioni dell'ordine di 2^{20}, ovverosia circa un milione di chiamate; calcolare il 30° numero di Fibonacci richiederà una quantità di invocazioni dell'ordine di 2^{30}, ovverosia un miliardo di chiamate e così via. Gli studiosi di informatica si riferiscono a ciò con l'espressione *complessità esponenziale*. I problemi di questa natura sono in grado di mortificare anche il più potente dei computer! I problemi relativi alla complessità in generale e a quella esponenziale in particolare saranno discussi in dettaglio in "Algoritmi e Strutture Dati": un corso di livello superiore incluso nel programma di studi di informatica.

Obiettivo efficienza 5.4

Evitate i programmi ricorsivi nello stile di `fibonacci` *che producano una "esplosione" esponenziale di invocazioni.*

5.16 Ricorsione e iterazione

Nelle sezioni precedenti, abbiamo studiato due funzioni che possono essere facilmente implementate in modo ricorsivo o iterativo. In questa sezione confronteremo i due approcci e discuteremo le ragioni per le quali, in situazioni particolari, un programmatore potrà scegliere un approccio piuttosto che l'altro.

L'iterazione e la ricorsione sono basate entrambe su una struttura di controllo: l'iterazione utilizza una ripetizione; la ricorsione utilizza una selezione. L'iterazione e la ricorsione

richiedono entrambe una ripetizione: la prima utilizza esplicitamente una struttura di quel tipo; la seconda la ottiene invece attraverso ripetute chiamate di funzione. L'iterazione e la ricorsione richiedono entrambe un controllo di terminazione: l'iterazione terminerà quando sarà fallita la condizione di continuazione del ciclo; la ricorsione terminerà invece quando sarà stato riconosciuto un caso di base. L'iterazione con una ripetizione controllata da un contatore e la ricorsione si approssimeranno gradualmente alla conclusione: l'iterazione continuerà a modificare un contatore finché questo assumerà un valore che farà fallire la condizione di continuazione del ciclo; la ricorsione continuerà a produrre versioni più semplici del problema originario finché non sarà stato ottenuto un caso di base. L'iterazione e la ricorsione potranno entrambe ripetersi all'infinito: con l'iterazione otterremo un ciclo infinito qualora il relativo controllo di continuazione non diventi mai falso; otterremo una ricorsione infinita qualora il passo di ricorsione non riduca ogni volta il problema in modo da convergere verso un caso di base.

La ricorsione ha molti aspetti negativi. Coinvolge ripetutamente il meccanismo, e di conseguenza il peso, di una invocazione di funzione. Ciò potrà essere oneroso in termini di tempo del processore e di spazio di memoria. Ogni chiamata ricorsiva provocherà la creazione di un'altra copia della funzione (in realtà solo delle sue variabili); ciò potrà consumare una considerevole quantità di memoria. L'iterazione invece normalmente avverrà all'interno di una funzione, perciò sarà evitato il peso dovuto alle ripetute chiamate di funzione e all'assegnamento di memoria aggiuntiva. Pertanto perché scegliere la ricorsione?

Ingegneria del software 5.13

Qualsiasi problema che possa essere risolto in modo ricorsivo potrà anche essere risolto in maniera iterativa (non ricorsiva). Normalmente l'approccio ricorsivo sarà preferito a quello iterativo, qualora il primo rispecchi in modo più naturale il problema e produca un programma che sia più facile da comprendere e collaudare. Un'altra ragione per scegliere una soluzione ricorsiva è che quella iterativa potrebbe non essere evidente.

Obiettivo efficienza 5.5

Evitate di usare le ricorsioni in situazioni in cui l'efficienza sia un aspetto critico. Le chiamate ricorsive richiedono tempo e consumano più memoria.

Errore tipico 5.18

Fare in modo che una funzione non ricorsiva richiami accidentalmente se stessa in modo diretto o indiretto attraverso un'altra funzione.

La maggior parte dei libri di testo sulla programmazione affrontano la ricorsione più tardi di quanto abbiamo fatto in questo libro. Noi crediamo che la ricorsione sia un argomento così ricco e complesso che è preferibile affrontare al più presto, disseminando il resto del libro con degli esempi. La Figura 5.17 riassume per capitoli i 31 esempi ed esercizi di ricorsione presenti in questo testo.

Lasciateci chiudere questo capitolo con alcune osservazioni che abbiamo formulato ripetutamente nel corso del libro. Una buona progettazione del software è importante. Anche l'efficienza è importante. Sfortunatamente, questi obiettivi sono spesso in contrasto l'uno con l'altro. Una buona progettazione del software è di importanza vitale, per rendere più maneggevole lo sviluppo dei sistemi software più corposi e complessi di cui abbiamo bisogno. L'efficienza è di importanza vitale, per realizzare i sistemi del futuro che sottoporranno al-

l'hardware delle esigenze di elaborazione sempre maggiori. Quale sarà il ruolo delle funzioni in questa situazione?

Capitolo	Esempi ed esercizi di ricorsione
Capitolo 5	Funzione fattoriale Funzione di Fibonacci Massimo comun divisore Somma di due interi Moltiplicazione di due interi Elevamento di un intero a una potenza intera Le torri di Hanoi La funzione `main` ricorsiva Visualizzare al contrario l'input della tastiera Visualizzare la ricorsione
Capitolo 6	Somma degli elementi di un vettore Visualizzare un vettore Visualizzare al contrario un vettore Visualizzare al contrario una stringa Verificare se una stringa sia palindroma Valore minimo di un vettore Ordinamento per selezione Quicksort Ricerca lineare Ricerca binaria Le otto Regine
Capitolo 7	Attraversamento di un labirinto
Capitolo 8	Visualizzare al contrario una stringa immessa dalla tastiera
Capitolo 12	Inserimento in una lista concatenata Eliminazione da una lista concatenata Ricerca in una lista concatenata Visualizzare al contrario una lista concatenata Inserimento in un albero binario Visita anticipata di un albero binario Visita in ordine (simmetrica) di un albero binario Visita differita di un albero binario

Figura 5.17 Sommario degli esempi e degli esercizi sulla ricorsione presenti in questo testo

Obiettivo efficienza 5.6

Suddividere in funzioni un programma in modo chiaro e gerarchico favorisce una buona progettazione del software. Tuttavia, ciò ha un prezzo. Un programma eccessivamente suddiviso in funzioni, rispetto a uno monolitico (ovverosia in un unico pezzo) che non le abbia, eseguirà potenzialmente un gran numero di chiamate di funzioni e ciò consumerà il tempo di esecuzione dei processori di un computer. Quindi, nonostante i programmi monolitici possano avere delle prestazioni migliori, essi saranno difficili da programmare, collaudare, mettere a punto, mantenere e far evolvere.

Suddividete quindi il vostro programma in funzioni con giudizio, tenendo sempre a mente il delicato equilibrio tra l'efficienza e la buona progettazione del software.

Esercizi di autovalutazione

5.1 Rispondete a ognuna delle seguenti domande:

a) Un modulo di programma in C è una _____.
b) Una funzione è invocata con una _____.
c) Una variabile che sia nota solo all'interno della funzione in cui è stata definita è una _____.
d) L'istruzione _____ di una funzione chiamata sarà utilizzata per restituire a quella chiamante il valore di una espressione.
e) La parola chiave _____ sarà utilizzata nell'intestazione di una funzione, per indicare che non restituirà un valore o che non conterrà dei parametri.
f) La _____ di un identificatore è la porzione del programma in cui quello potrà essere utilizzato.
g) I tre modi per restituire il controllo da una funzione chiamata al chiamante sono _____, _____ e _____.
h) Un _____ consente al compilatore di verificare il numero, i tipi e l'ordine degli argomenti passati a una funzione.
i) La funzione _____ è utilizzata per generare dei numeri casuali.
j) La funzione _____ è utilizzata per impostare il seme del numero casuale, in modo da randomizzare un programma.
k) Le specifiche di classe di memoria sono _____, _____, _____ e _____.
l) Le variabili dichiarate in un blocco o nella lista dei parametri di una funzione, sempre che non sia stato specificato diversamente, saranno considerate della classe di memoria _____.
m) La specifica di classe di memoria _____ è una raccomandazione rivolta al compilatore affinché immagazzini una variabile in uno dei registri del computer.
n) Una variabile dichiarata all'esterno di qualsiasi blocco o funzione è una variabile _____.
o) Perché una variabile locale di una funzione possa conservare il proprio valore tra le diverse chiamate alla funzione stessa, essa dovrà essere dichiarata con la specifica di classe di memoria _____.
p) Le quattro possibili visibilità per un identificatore sono _____, _____, _____ e _____.
q) Una funzione che richiama se stessa in modo diretto o indiretto è una funzione _____.
r) Una funzione ricorsiva avrà tipicamente due parti: una che fornisca un mezzo per terminare la ricorsione, verificando l'occorrenza di un caso _____, e l'altra che esprima il problema con una chiamata ricorsiva a uno leggermente più semplice di quello della chiamata originaria.

5.2 Per il seguente programma, stabilite la visibilità (nella funzione, nel file, nel blocco o nel prototipo di funzione) di ognuno dei seguenti elementi.

a) La variabile `x` nella funzione `main`.
b) La variabile `y` in `cube`.
c) La funzione `cube`.
d) La funzione `main`.
e) Il prototipo di funzione per `cube`.
f) L'identificatore `y` nel prototipo di funzione di `cube`.

```
1   #include <stdio.h>
2   int cube( int y );
3
4   int main()
5   {
6         int x;
7
8         for ( x = 1; x <= 10; x++ )
9               printf( "%d\n", cube( x ) );
10        return 0 ;
11  }
12
13  int cube( int y )
14  {
15        return y * y * y;
16  }
```

5.3 Scrivete un programma che controlli se gli esempi di chiamate a funzioni della libreria matematica mostrati nella Figura 5.2 producano davvero i risultati indicati.

5.4 Fornite l'intestazione per ognuna delle seguenti funzioni.

a) La funzione `hypotenuse` che riceva due argomenti in virgola mobile a doppia precisione, `side1` e `side2`, e restituisca un risultato in virgola mobile a doppia precisione.
b) La funzione `smallest` che prenda tre interi, `x`, `y` e `z`, e restituisca un intero.
c) La funzione `instructions` che non riceva alcun argomento, né restituisca un valore. [*Nota*: tali funzioni sono utilizzate per visualizzare delle istruzioni per l'utente.]
d) La funzione `intToFloat` che prenda un argomento intero, `number`, e restituisca un risultato in virgola mobile.

5.5 Fornite il prototipo di funzione per ognuna delle seguenti funzioni:

a) La funzione descritta nell'Esercizio 5.4a.
b) La funzione descritta nell'Esercizio 5.4b.
c) La funzione descritta nell'Esercizio 5.4c.
d) La funzione descritta nell'Esercizio 5.4d.

5.6 Scrivete una dichiarazione per ognuna delle seguenti variabili:

a) L'intero `count` che dovrebbe essere immagazzinato in un registro. Inizializzate `count` a `0`.
b) La variabile in virgola mobile `lastVal` che dovrà conservare il proprio valore tra le chiamate alla funzione in cui verrà definita.
c) L'intero esterno `number` la cui visibilità dovrà essere limitata al resto del file in cui verrà definito.

5.7 Trovate l'errore in ognuno dei seguenti segmenti di programma e spiegate come potrà essere corretto (consultate anche l'Esercizio 5.50):

a)
```
int g( void )
{
   printf( "Inside function g\n" );
   int h( void )
   {
         printf( "Inside function h\n" );
   }
}
```

b)
```
int sum( int x, int y )
{
   int result;
   result = x + y;
```

```
   }
c) int sum( int n )
   {
      if ( n == 0 )
            return 0;
      else
            n + sum( n - 1 );
   }
d) void f( float a );
   {
      float a;
   printf( "%f", a );
}
e) void product( void )
      {
      int a, b, c, result;
      printf( "Enter three integers: " )
      scanf( %d%d%d", &a, &b, &c );
      result = a * b * c;
      printf( "Result is %d", result );
      return result;
   }
```

Risposte agli esercizi di autovalutazione

5.1 a) Funzione. b) Chiamata di funzione. c) Variabile locale. d) `return`. e) `void`. f) Visibilità. g) `return;` o `return espressione;` o incontrando la parentesi graffa di chiusura di una funzione. h) Prototipo di funzione. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) `register`. n) Esterna, globale. o) `static`. p) Visibilità nella funzione, visibilità nel file, visibilità nel blocco, visibilità nel prototipo di funzione. q) Ricorsiva. r) Di base.

5.2 a) Visibilità nel blocco. b) Visibilità nel blocco. c) Visibilità nel file. d) Visibilità nel file. e) Visibilità nel file. f) Visibilità nel prototipo di funzione.

5.3

```
 1  /* ex05_03.c */
 2  /* Verificare le funzioni della libreria matematica */
 3  #include <stdio.h>
 4  #include <math.h>
 5
 6  /* l'esecuzione del programma inizia dalla funzione main */
 7  int main()
 8  {
 9     /* calcola e visualizza la radice quadrata */
10     printf( "sqrt(%.1f) = %.1f\n", 900.0, sqrt( 900.0 ) );
11     printf( "sqrt(%.1f) = %.1f\n", 9.0, sqrt( 9.0 ) );
12
13     /* calcola e visualizza la funzione esponenziale e elevata alla x */
14     printf( "exp(%.1f) = %f\n", 1.0, exp( 1.0 ) );
15     printf( "exp(%.1f) = %f\n", 2.0, exp( 2.0 ) );
16
17     /* calcola e visualizza il logaritmo (base e) */
18     printf( "log(%f) = %.1f\n", 2.718282, log( 2.718282 ) );
19     printf( "log(%f) = %.1f\n", 7.389056, log( 7.389056 ) );
20
21     /* calcola e visualizza il logaritmo (base 10) */
```

```
22      printf( "log10(%.1f) = %.1f\n", 1.0, log10( 1.0 ) );
23      printf( "log10(%.1f) = %.1f\n", 10.0, log10( 10.0 ) );
24      printf("log10(%.1f) = %.1f\n", 100.0, log10( 100.0 ) );
25
26      /* calcola e visualizza il valore assoluto */
27      printf( "fabs(%.1f) = %.1f\n", 13.5, fabs( 13.5 ) );
28      printf( "fabs(%.1f) = %.1f\n", 0.0, fabs( 0.0 ) );
29      printf( "fabs(%.1f) = %.1f\n", -13.5, fabs( -13.5 ) );
30
31      /* calcola e visualizza ceil( x ) */
32      printf( "ceil(%.1f) = %.1f\n", 9.2, ceil( 9.2 ) );
33      printf( "ceil(%.1f) = %.1f\n", -9.8, ceil( -9.8 ) );
34
35      /* calcola e visualizza floor( x ) */
36      printf( "floor(%.1f) = %.1f\n", 9.2, floor( 9.2 ) );
37      printf( "floor(%.1f) = %.1f\n", -9.8, floor( -9.8 ) );
38
39      /* calcola e visualizza pow( x, y ) */
40      printf( "pow(%.1f, %.1f) = %.1f\n", 2.0, 7.0, pow( 2.0, 7.0 ) );
41      printf( "pow(%.1f, %.1f) = %.1f\n", 9.0, 0.5, pow( 9.0, 0.5 ) );
42
43      /* calcola e visualizza fmod( x, y ) */
44      printf( "fmod(%.3f/%.3f) = %.3f\n", 13.675, 2.333,
45          fmod(13.675, 2.333));
46
47      /* calcola e visualizza sin( x ) */
48      printf( "sin(%.1f) = %.1f\n", 0.0, sin( 0.0 ) );
49
50      /* calcola e visualizza cos( x ) */
51      printf( "cos(%.1f) = %.1f\n", 0.0, cos( 0.0 ) );
52
53      /* calcola e visualizza tan( x ) */
54      printf( "tan(%.1f) = %.1f\n", 0.0, tan( 0.0 ) );
55
56      return 0; /* indica che il programma è terminato con successo */
57
58  } /* fine della funzione main */
```

```
sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
```

```
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

5.4 a) `double hypotenuse( double side1, double side2 )`
b) `int smallest( int x, int y, int z )`
c) `void instructions( void )`
d) `float intToFloat( int number )`

5.5 a) `double hypotenuse( double side1, double side2 );`
b) `int smallest( int x, int y, int z );`
c) `void instructions( void );`
d) `float intToFloat( int number );`

5.6 a) `register int count = 0;`
b) `static float lastVal;`
c) `static int number;`

[*Nota*: l'ultima dichiarazione dovrà comparire all'esterno di ogni definizione di funzione.]

5.7 a) Errore: la funzione `h` è stata definita all'interno della funzione `g`.
Correzione: spostate la definizione di `h` all'esterno della definizione di `g`.
b) Errore: il corpo della funzione dovrebbe restituire un intero, ma non lo fa.
Correzione: eliminate la variabile `result` e inserite nella funzione l'istruzione seguente:

```
return x + y;
```

c) Errore: il risultato di `n + sum( n - 1 )` non è stato restituito; `sum` restituirà un risultato inappropriato.
Correzione: riscrivete l'istruzione nella clausola `else` come

```
return n + sum( n - 1 );
```

d) Errore: il punto e virgola dopo la parentesi destra che chiude la lista dei parametri e la ridefinizione del parametro `a` nel corpo della funzione.
Correzione: eliminate il punto e virgola dopo la parentesi destra della lista dei parametri, ed eliminate la dichiarazione `float a;`.
e) Errore: la funzione restituisce un valore e non dovrebbe farlo.
Correzione: eliminate l'istruzione `return`.

Esercizi

5.8 Mostrate il valore di `x` dopo che sarà stata eseguita ognuna delle seguenti istruzioni:
a) `x = fabs( 7.5 );`
b) `x = floor( 7.5 );`
c) `x = fabs( 0.0 );`
d) `x = ceil( 0.0 ) ;`
e) `x = fabs( -6.4 );`
f) `x = ceil( -6.4 );`
g) `x = ceil( -fabs( -8+floor( -5.5) ) );`

5.9 Un garage addebita un importo minimo di $ 2,00 per un parcheggio fino a tre ore. Il garage addebita un'addizionale di $ 0,50 per ogni ora o frazione di essa che ecceda le tre di base. L'addebito

massimo per ogni dato periodo di 24 ore è $ 10,00. Assumete che nessuna auto parcheggi per più di 24 ore per volta. Scrivete un programma che calcoli e visualizzi gli addebiti per ognuno dei tre clienti che hanno parcheggiato le proprie auto in questo garage ieri. Dovrete immettere le ore di parcheggio per ogni cliente. Il vostro programma dovrà visualizzare i risultati in un formato tabulare ordinato e dovrà calcolare e visualizzare il totale delle ricevute di ieri. Il programma dovrà utilizzare la funzione `calculateCharges` per determinare l'addebito di ogni cliente. I vostri risultati dovranno apparire nel seguente formato:

```
Car     Hours     Charge
1         1.5       2.00
2         4.0       2.50
3        24.0      10.00
TOTAL    29.5      14.50
```

5.10 Un'applicazione della funzione `floor` è l'arrotondamento di un valore all'intero più vicino. L'istruzione

```
y = floor( x + .5 );
```

arrotonderà il numero `x` all'intero più vicino e assegnerà il risultato a `y`. Scrivete un programma che legga diversi numeri e utilizzi l'istruzione precedente per arrotondare ognuno di questi numeri all'intero più vicino. Per ogni numero elaborato, visualizzate quello originale e quello arrotondato.

5.11 La funzione `floor` può essere utilizzata per arrotondare un numero a una specifica posizione decimale. L'istruzione

```
y = floor( x * 10 + .5 ) / 10;
```

arrotonda `x` alla posizione dei decimi (la prima a destra della virgola dei decimali). L'istruzione

```
y = floor( x * 100 + .5 ) / 100;
```

arrotonda `x` alla posizione dei centesimi (la seconda a destra della virgola dei decimali). Scrivete un programma che definisca quattro funzioni per arrotondare un numero `x` in vari modi

a) `roundToInteger( number )`
b) `roundToTenths( number )`
c) `roundToHundreths( number )`
d) `roundToThousandths( number )`

Per ogni valore letto, il vostro programma dovrà visualizzare il numero originale e quelli arrotondati all'intero, al decimo, al centesimo e al millesimo più vicini.

5.12 Rispondete a ognuna delle seguenti domande.

a) Cosa significa scegliere dei numeri "a caso"?
b) Perché la funzione `rand` è utile nella simulazione dei giochi d'azzardo?
c) Perché dovreste randomizzare un programma, utilizzando `srand`? In quali circostanze è preferibile non randomizzare?
d) Perché è spesso necessario ridurre in scala e/o traslare i valori prodotti da `rand`?
e) Perché la simulazione computerizzata di situazioni del mondo reale è una tecnica utile?

5.13 Scrivete delle istruzioni che assegnino alla variabile n degli interi casuali compresi nei seguenti intervalli:

a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$

e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

5.14 Per ognuno dei seguenti gruppi di interi, scrivete una singola istruzione che visualizzi un numero casuale tratto dal gruppo.

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

5.15 Definite una funzione `hypotenuse` che calcoli la lunghezza dell'ipotenusa di un triangolo rettangolo quando siano dati gli altri due lati. Utilizzate questa funzione in un programma che determini la lunghezza dell'ipotenusa per ognuno dei seguenti triangoli. La funzione dovrà ricevere due argomenti di tipo `double` e restituire l'ipotenusa come un `double`. Testate il vostro programma con i valori dei lati specificati nella Figura 5.18.

Triangolo	Lato 1	Lato 2
1	3,0	4,0
2	5,0	12,0
3	8,0	15,0

Figura 5.18 Campioni di valori di lati di triangoli per l'Esercizio 5.15

5.16 Scrivete una funzione `integerPower( base, exponent )` che restituisca il valore di

$base^{exponent}$

Per esempio, `integerPower( 3, 4 ) = 3 * 3 * 3 * 3`. Supponete che `exponent` sia un intero positivo diverso da zero e che base sia un intero. La funzione `integerPower` dovrà utilizzare `for` per controllare il calcolo. Non utilizzate nessuna funzione della libreria matematica.

5.17 Scrivete una funziona `multiple` che per una coppia di interi determini se il secondo sia un multiplo del primo. La funzione dovrà ricevere due argomenti interi e restituire `1` (vero), qualora il secondo sia un multiplo del primo, `0` (falso), in caso contrario. Utilizzate questa funzione in un programma che prenda in input una serie di coppie di interi.

5.18 Scrivete un programma che prenda in input una serie di interi e li passi, uno per volta, alla funzione `even`, che utilizzerà l'operatore modulo per determinare se un intero è pari. La funzione dovrà ricevere un argomento intero e restituire `1` qualora l'intero sia pari e `0`, in caso contrario.

5.19 Scrivete una funzione che visualizzi al margine sinistro dello schermo un quadrato pieno di asterischi il cui lato sia stato specificato nel parametro intero `side`. Per esempio, qualora `side` fosse `4`, la funzione dovrebbe visualizzare:

```
****
****
****
****
```

5.20 Modificate la funzione creata nell'Esercizio 5.19 in modo da formare il quadrato con qualsiasi carattere sia contenuto nel parametro di tipo carattere `fillCharacter`. Di conseguenza, qualora `side` fosse `5` e `fillCharacter` fosse "#", la funzione dovrebbe visualizzare:

```
#####
#####
#####
#####
#####
```

5.21 Utilizzate delle tecniche simili a quelle sviluppate negli Esercizi 5.19 e 5.20, per produrre un programma che tracci un'ampia gamma di forme.

5.22 Scrivete dei segmenti di programma che eseguano ognuno dei seguenti compiti:

a) Calcolate la parte intera del quoziente ottenuto dalla divisione degli interi `a` e `b`.
b) Calcolate il resto intero ottenuto dalla divisione degli interi `a` e `b`.
c) Utilizzate i pezzi di programma sviluppati in a) e b) per scrivere una funzione che prenda in input un intero, compreso tra `1` e `32767`, e lo visualizzi come una sequenza di cifre, separando ogni coppia di esse con due spazi. Per esempio, l'intero `4562` dovrà essere stampato come:

```
4 5 6 2
```

5.23 Scrivete una funzione che accetti in input l'ora, suddivisa in tre argomenti interi (per le ore, i minuti e i secondi), e restituisca il numero dei secondi trascorsi dall'ultima volta che l'orologio "ha rintoccato le 12". Utilizzate questa funzione per calcolare la quantità di tempo in secondi che intercorre tra due orari, entrambi i quali siano compresi all'interno di un ciclo di 12 ore dell'orologio.

5.24 Implementate le seguenti funzioni intere:

a) La funzione `celsius` restituisce l'equivalente Celsius di una temperatura Fahrenheit (utilizzate la formula °C=(5/9) x (°F-32), dove °C rappresenta la temperatura in gradi Celsius e °F quella in gradi Fahrenheit).
b) La funzione `fahrenheit` restituisce l'equivalente Fahrenheit di una temperatura Celsius (utilizzate la formula °F=(9/5)°C+32, dove °C rappresenta la temperatura in gradi Celsius e °F quella in gradi Fahrenheit).
c) Utilizzate queste funzioni, per scrivere un programma che visualizzi delle tabelle che mostrino gli equivalenti Fahrenheit di tutte le temperature Celsius comprese tra 0 e 100 gradi, nonché gli equivalenti Celsius di tutte le temperature Fahrenheit comprese tra 32 e 212 gradi. Visualizzate i risultati in una forma tabulare ordinata che minimizzi il numero di righe dell'output, pur rimanendo leggibile.

5.25 Scrivete una funzione che restituisca il minore di tre numeri in virgola mobile.

5.26 Un numero intero è detto numero perfetto qualora la somma dei suoi fattori, incluso 1 (ma non se stesso), sia pari a quel numero. Per esempio 6 è un numero perfetto perché 6 = 1 + 2 + 3. Scrivete una funzione `perfect` che determini se il parametro `number` sia un numero perfetto. Utilizzate questa funzione in un programma che determini e visualizzi tutti i numeri perfetti tra 1 e 1000. Visualizzate i fattori di ogni numero perfetto, per confermare che lo sia veramente. Sfidate la potenza del vostro computer, provando con numeri maggiori di 1000.

5.27 Un intero è detto primo qualora sia divisibile soltanto per 1 e per se stesso. Per esempio 2, 3, 5 e 7 sono primi, mentre 4, 6, 8 e 9 non lo sono.

a) Scrivete una funzione che determini se un numero sia primo.
b) Utilizzate questa funzione in un programma che determini e visualizzi tutti i numeri primi tra 1 e 10.000. Quanti di questi 10.000 numeri dovrete realmente verificare prima di essere sicuri di aver trovato tutti i numeri primi?
c) Inizialmente potreste pensare che $n/2$ sia il limite superiore che dovrete verificare per vedere se un numero sia primo, ma in realtà vi basterà arrivare alla radice quadrata di n. Perché?

Riscrivete il programma e fatelo eseguire in entrambi i modi. Stimate il miglioramento delle prestazioni.

5.28 Scrivete una funzione che prenda un valore intero e lo restituisca dopo avere invertito le sue cifre. Per esempio, dato il numero 7631, la funzione dovrà restituire 1367.

5.29 Il massimo comun divisore (MCD) di due interi è l'intero più grande che possa dividere esattamente ognuno dei due numeri. Scrivete una funzione `gcd` che restituisca il massimo comun divisore di due interi.

5.30 Scrivete una funzione `qualityPoints` che prenda in input la media di uno studente e restituisca 4 qualora la sua media sia compresa tra 90 e 100, 3 tra 80 e 89, 2 tra 70 e 79, 1 tra 60 e 69 e 0 qualora sia inferiore a 60.

5.31 Scrivete un programma che simuli il lancio di una monetina. Per ogni lancio della monetina il programma dovrà visualizzare `Heads` o `Tails`. Lasciate che il programma lanci la monetina per 100 volte e contate il numero di occorrenze per la comparsa di ogni faccia della monetina. Visualizzate i risultati. Il programma dovrà chiamare una funzione distinta `flip`, che non riceverà argomenti e che restituirà `0` per croce e `1` per testa. [*Nota*: qualora il programma simuli realisticamente il lancio di una monetina, allora ogni faccia della stessa dovrà apparire approssimativamente la metà delle volte, per un totale approssimativo di 50 teste e 50 croci.]

5.32 I computer giocano un ruolo sempre più importante nell'educazione. Scrivete un programma che aiuti uno studente di scuola elementare ad apprendere la moltiplicazione. Utilizzate `rand` per produrre due interi positivi di una cifra. Dovrete quindi visualizzare una domanda come:

```
How much is 6 times 7?
```

In seguito lo studente digiterà la risposta. Il vostro programma controllerà la risposta dello studente. Qualora sia corretta, visualizzate `"Very good!"` e sottoponete quindi un'altra domanda sulla moltiplicazione. Nel caso che la risposta sia sbagliata, visualizzate `"No. Please try again."` e lasciate quindi che lo studente provi ancora ripetutamente la stessa domanda, finché alla fine non avrà risposto correttamente.

5.33 L'utilizzo dei computer nell'educazione è detta istruzione assistita dal computer (CAI, computer-assisted instruction). Uno dei problemi che si sviluppa negli ambienti CAI è l'affaticamento dello studente. Questo può essere eliminato, variando il dialogo del computer in modo da mantenere viva l'attenzione dello studente. Modificate il programma dell'Esercizio 5.32 in modo che siano visualizzati vari commenti a fronte di ogni risposta corretta e sbagliata, come segue:

Commenti per una risposta corretta

```
Very good!
Excellent!
Nice work!
Keep up the good work!
```

Commenti per una risposta sbagliata

```
No. Please try again.
Wrong. Try once more.
Don't give up!
No. Keep trying.
```

Utilizzate il generatore di numeri casuali per sceglierne uno compreso tra 1 a 4 e selezionare un commento appropriato a ogni risposta. Utilizzate la struttura `switch` con delle istruzioni `printf` per emettere i commenti.

5.34 I sistemi più raffinati di istruzione assistita dal computer monitorano le prestazioni dell'utente durante un certo periodo. La decisione di incominciare un nuovo argomento sarà spesso basata sul successo dello studente con gli argomenti precedenti. Modificate il programma dell'Esercizio 5.33 in modo da contare il numero di risposte corrette e sbagliate immesse dallo studente. Dopo che lo studente avrà digitato 10 risposte, il vostro programma dovrà calcolare la percentuale di quelle corrette. Qualora la percentuale sia inferiore al 75 percento, il vostro programma dovrà visualizzare `"Please ask your instructor for extra help"` e quindi terminare la propria esecuzione.

5.35 Scrivete un programma C che proponga il gioco "indovina il numero" nel modo seguente: il vostro programma sceglierà il numero da indovinare, selezionando un intero a caso compreso nell'intervallo da 1 a 1000. Il programma quindi visualizzerà:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
```

Il giocatore allora digiterà una prima ipotesi. Il programma risponderà con una delle seguenti frasi:

```
1. Excellent! You guessed the number!
   Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.
```

Qualora l'ipotesi del giocatore sia sbagliata, il vostro programma dovrà reiterare finché il giocatore non avrà finalmente indovinato il numero corretto. Il vostro programma dovrà continuare a indicare `Too high` o `Too low`, per aiutare il giocatore a "prendere la mira" sulla risposta corretta. [*Nota*: la tecnica di ricerca impiegata in questo problema è detta ricerca binaria. Diremo qualcosa in più su ciò nel prossimo problema.]

5.36 Modificate l'Esercizio 5.35 in modo da contare il numero di tentativi del giocatore. Qualora il numero sia inferiore o uguale a 10, visualizzate "`Either you know the secret or you got lucky!`". Qualora il giocatore indovini il numero in 10 tentativi, visualizzate "`Ahah! You know the secret!`". Qualora il giocatore faccia più di 10 tentativi, visualizzate "`You should be able to do better!`". Per quale motivo non dovranno essere consentiti più di 10 tentativi? Ebbene con ogni "buona ipotesi" il giocatore dovrebbe essere in grado di eliminare la metà dei numeri. Dimostrate ora in che modo ogni numero tra 1 e 1000 possa essere indovinato in 10 tentativi o meno.

5.37 Scrivete una funzione ricorsiva `power ( base, exponent )` che quando invocata restituisca

$$base^{exponent}$$

Per esempio, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Supponete che `exponent` sia un intero maggiore o uguale a `1`. Suggerimento: il passo di ricorsione dovrà utilizzare la relazione

$$base^{exponent} = base \cdot base^{exponent-1}$$

e la condizione di terminazione sarà verificata quando `exponent` sarà uguale a `1` perché

$$base^{1} = base$$

5.38 La serie di Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

incomincia con i termini 0 e 1 e gode della proprietà per la quale ogni temine successivo sarà pari alla somma dei due termini precedenti. a) Scrivete una funzione `fibonacci( n )` non ricorsiva che calcola il numero `n`-esimo di Fibonacci. b) Determinate il numero di Fibonacci più grande che possa essere

visualizzato sul vostro sistema. Modificate il programma della parte a) in modo da usare dei `double`, invece degli `int`, per calcolare e restituire i numeri di Fibonacci. Lasciate reiterare il programma finché non fallisca a causa di un valore eccessivamente alto.

5.39 (*Le torri di Hanoi*) Ogni informatico in erba dovrà venire alle prese con certi problemi classici e le torri di Hanoi (consultate la Figura 5.19) è uno dei più famosi di questi. La leggenda narra che, in un tempio dell'Estremo Oriente, alcuni preti stavano tentando di muovere una pila di dischi da un paletto a un altro. La pila iniziale aveva 64 dischi infilati su un paletto e ordinati in misura decrescente dal basso all'alto. I preti stavano tentando di muovere la pila da questo a un secondo paletto, rispettando delle regole secondo le quali: poteva essere mosso esattamente un disco per volta e in nessun momento un disco più grande poteva essere sistemato su uno più piccolo. Era disponibile un terzo paletto per riporre temporaneamente i dischi. Secondo la leggenda, qualora i preti avessero completato il loro compito, sarebbe giunta la fine del mondo, perciò non abbiamo molto interesse nel facilitare i loro sforzi.

Supponiamo che i preti stiano tentando di muovere i dischi dal paletto 1 al 3. Vogliamo sviluppare un algoritmo che visualizzi l'esatta sequenza di trasferimenti di paletto disco per disco.

Se dovessimo affrontare questo problema con i metodi convenzionali, ci ritroveremmo presto disperatamente ingarbugliati nella manipolazione dei dischi. Se invece affrontiamo il problema tenendo in mente la ricorsione, esso diventerà immediatamente trattabile. Lo spostamento di n dischi potrà essere visto come lo spostamento di soli $n - 1$ dischi (e quindi la ricorsione) come segue:

1. Muovere $n - 1$ dischi dal paletto 1 al 2, utilizzando il 3 come area di deposito temporanea.
2. Muovere l'ultimo disco (il più grande) dal paletto 1 al 3.
3. Muovere gli $n - 1$ dischi dal paletto 2 al 3, utilizzando il paletto 1 come area di deposito temporanea.

Il processo terminerà quando l'ultimo compito richiederà lo spostamento di $n = 1$ dischi, ovverosia il caso di base. Ciò sarà eseguito semplicemente spostando il disco, senza la necessità di un'area di deposito temporanea.

Scrivete un programma che risolva il problema delle torri di Hanoi. Utilizzate una funzione ricorsiva con quattro parametri:

1. Il numero dei dischi da muovere.
2. Il paletto sul quale questi dischi saranno inizialmente infilati.
3. Il paletto sul quale questa pila di dischi dovrà essere spostata.
4. Il paletto che dovrà essere utilizzato come area di deposito temporanea.

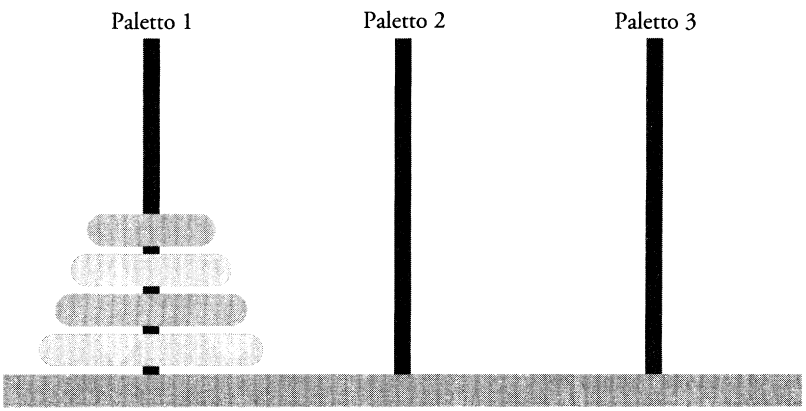


Figura 5.19 Le torri di Hanoi per il caso con quattro dischi

Il vostro programma dovrà visualizzare le istruzioni precise necessarie per lo spostamento dei dischi dal paletto di partenza a quello di destinazione. Per esempio, per muovere una pila di tre dischi dal paletto 1 al 3, il vostro programma dovrà visualizzare la seguente serie di mosse:

1 → 3 (Questo significa muovere un disco dal paletto 1 al paletto 3).
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

5.40 Ogni programma che possa essere implementato in modo ricorsivo potrà essere costruito in maniera iterativa, sebbene a volte con maggiori difficoltà e minor chiarezza. Cercate di scrivere una versione iterativa delle torri di Hanoi. Qualora vi riusciste, confrontate la vostra versione iterativa con quella ricorsiva che avete sviluppato nell'Esercizio 5.39. Analizzate i problemi di efficienza, di chiarezza e la vostra capacità nel dimostrare la correttezza dei programmi.

5.41 (*Visualizzare la ricorsione*) È interessante osservare la ricorsione "in azione". Modificate la funzione del fattoriale della Figura 5.14 in modo da visualizzare le sue variabili locali e i parametri delle chiamate ricorsive. Per ogni chiamata ricorsiva, visualizzate gli output su una riga separata e aggiungete un livello di rientro. Fate del vostro meglio per rendere l'output chiaro, interessante e significativo. Il vostro obiettivo in questo caso sarà di disegnare e implementare un formato dell'output che aiuti una persona a capire meglio la ricorsione. Potreste voler aggiungere queste capacità di visualizzazione ai molti altri esempi ed esercizi sulla ricorsione presenti in questo testo.

5.42 Il massimo comun divisore degli interi `x` e `y` è l'intero più grande che divida esattamente sia `x` che `y`. Scrivete una funzione ricorsiva `gcd` che restituisca il massimo comun divisore di `x` e `y`. Il `gcd` di `x` e `y` sarà definito in modo ricorsivo come segue: se `y` è uguale a `0`, allora `gcd( x, y )` sarà `x`; altrimenti `gcd( x, y )` sarà `gcd( y, x % y )` dove `%` sarà l'operatore resto.

5.43 Può il `main` essere chiamato in modo ricorsivo? Scrivete un programma che contenga una funzione `main`. Includete la variabile locale `static count` inizializzata a 1. Ogni volta che la funzione `main` è invocata, applicate l'operatore di postincremento e visualizzate il valore di `count`. Eseguite il programma. Che cosa succede?

5.44 Gli Esercizi dal 5.32 al 5.34 hanno permesso di sviluppare un programma di istruzione assistita dal computer, per insegnare la moltiplicazione a uno studente di scuola elementare. Questo esercizio suggerisce dei miglioramenti a quel programma.

a) Modificate il programma in modo da consentire all'utente di immettere un livello di difficoltà. Un livello 1 significherà utilizzare solo numeri di una cifra all'interno dei problemi, un livello 2 significherà utilizzare anche numeri di due cifre ecc.
b) Modificate il programma in modo da consentire all'utente di scegliere il tipo di problemi aritmetici che lui o lei desidera studiare. Un'opzione 1 significherà solo problemi sull'addizione, 2 solo sulla sottrazione, 3 solo sulla moltiplicazione, 4 solo sulla divisione mentre 5 significherà mischiare a caso problemi di tutti questi tipi.

5.45 Scrivete la funzione `distance` che calcoli la distanza tra due punti ($x1$, $y1$) e ($x2$, $y2$). Tutti i numeri e i valori restituiti dovranno essere di tipo `double`.

5.46 Che cosa farà il seguente programma?

```
1  #include <stdio.h>
2
3  /* l'esecuzione del programma inizia dalla funzione main */
4  int main()
```

```
5  {
6     int c; /* variabile per memorizzare il carattere inserito dall'utente */
7
8     if ( ( c = getchar() ) != EOF ) {
9        main();
10       printf( "%c", c );
11    } /* fine del comando if */
12
13    return 0; /* indica che il programma è terminato con successo */
14
15 } /* fine della funzione main */
```

5.47 Che cosa farà il seguente programma?

```
1  #include <stdio.h>
2
3  int mystery( int a, int b ); /* prototipo di funzione */
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int x; /* primo intero */
9     int y; /* secondo intero */
10
11    printf( "Enter two integers: " );
12    scanf( "%d%d", &x, &y );
13
14    printf( "The result is %d\n", mystery( x, y ) );
15
16    return 0; /* indica che il programma è terminato con successo */
17
18 } /* fine della funzione main */
19
20  /* Il parametro b deve essere un intero positivo
21     per prevenire una ricorsione infinita */
22  int mystery( int a, int b )
23 {
24     /* caso base */
25     if ( b == 1 ) {
26        return a;
27     } /* fine del ramo if */
28    else { /* passo ricorsivo */
29       return a + mystery( a, b - 1 );
30    } /* fine del ramo else */
31
32 } /* fine della funzione mystery */
```

5.48 Dopo che avrete determinato cosa farà il programma dell'Esercizio 5.47, modificatelo in modo che funzioni correttamente, dopo la rimozione della restrizione riguardante la positività del secondo parametro.

5.49 Scrivete un programma che controlli tante funzioni della libreria matematica della Figura 5.2 quante ne potete. Esercitatevi con ognuna di queste funzioni, facendo in modo che il vostro programma visualizzi delle tabelle contenenti i valori restituiti da una molteplicità di argomenti.

5.50 Trovate l'errore in ognuno dei seguenti segmenti di programma e spiegate come correggerlo:

a)
```
double cube( float );   /* prototipo di funzione */
...
```

```
   cube( float number )   /* definizione di funzione */
   {
      return number * number * number;
   }
```

b)
```
register auto int x = 7;
```

c)
```
int randomNumber = srand();
```

d)
```
double y = 123.45678;
int x;
x = y;
printf( "%f\n", (double) x );
```

e)
```
double square( double number )
{
   double number;

   return number * number;
}
```

f)
```
int sum( int n )
{
   if (n == 0)
       return 0;
   else
      return n + sum( n );
}
```

5.51 Modificate il programma per il gioco dei dadi della Figura 5.10 in modo da consentire delle scommesse. Impacchettate in una funzione la porzione del programma che eseguirà un singolo gioco ai dadi. Inizializzate la variabile `bankBalance` (saldo) a 1000 dollari. Chiedete al giocatore di immettere una `wager` (puntata). Utilizzate un ciclo `while` per controllare che `wager` sia inferiore o uguale a `bankBalance` e, in caso contrario, per chiedere all'utente di immettere nuovamente `wager` finché non ne sia stata immessa una valida. Dopo che sarà stata immessa una `wager` corretta, eseguite un gioco ai dadi. Nel caso in cui il giocatore vinca, incrementerete `bankBalance` di `wager` e visualizzerete il nuovo `bankBalance`. Nel caso in cui il giocatore perda, decrementerete `bankBalance` di `wager`, visualizzerete il nuovo `bankBalance`, verificherete che `bankBalance` non sia diventato zero e, in caso contrario, visualizzerete il messaggio `"Sorry, You busted!"`. Man mano che il gioco progredirà, visualizzerete vari messaggi per fare un po' di "chiacchiere" come `"Oh, you're going for broke, huh?"`, o `"Aw cmon, take a chance!"`, o `"You're up big. Now's the time to cash in your chips!"`.

CAPITOLO 6

I vettori in C

Obiettivi

- Presentare la struttura di dati vettore.
- Comprendere l'uso dei vettori per immagazzinare, ordinare e ricercare elenchi e tabelle di valori.
- Comprendere come dichiarare e inizializzare un vettore e come fare riferimento ai singoli elementi dello stesso.
- Essere in grado di passare i vettori alle funzioni.
- Comprendere le tecniche fondamentali di ordinamento.
- Essere in grado di dichiarare e manipolare dei vettori multidimensionali.

6.1 Introduzione

Questo capitolo fa da introduzione all'importante argomento delle strutture di dati. I *vettori* sono delle strutture di dati che consistono di unità di informazione dello stesso tipo. Nel Capitolo 10 discuteremo la nozione di `struct` (struttura) nel C: una struttura di dati formata da unità di informazione correlate ma con tipi eventualmente differenti. I vettori e le strutture sono "entità" statiche, giacché manterranno le proprie dimensioni durante l'esecuzione del programma. Naturalmente, essi potranno appartenere alla classe di memoria automatica ed essere quindi creati e distrutti ogni volta che si entrerà e si uscirà dai blocchi in cui saranno stati definiti. Nel Capitolo 12 introdurremo le strutture di dati dinamiche come le liste, le code, le pile e gli alberi che potranno crescere e decrescere durante l'esecuzione del programma.

6.2 I vettori

Un vettore è un gruppo di posizioni (o locazioni) di memoria correlate dal fatto che tutte hanno lo stesso nome e tipo di dato. Per far riferimento a una particolare posizione o elemento all'interno del vettore, specificheremo il nome dello stesso e il *numero di posizione* di quel particolare elemento nel vettore.

La Figura 6.1 mostra un vettore di interi chiamato `c`. Questo vettore contiene dodici *elementi*. Ognuno di questi elementi potrà essere puntato con il nome del vettore, seguito dal numero di posizione per quel particolare elemento racchiuso tra parentesi quadre (`[]`). Il primo elemento di ogni vettore è l'*elemento zero*. Di conseguenza, il primo elemento del vettore `c` sarà puntato da `c[ 0 ]`, il secondo elemento del vettore `c` sarà puntato da `c[ 1 ]`, il settimo elemento del vettore `c` sarà puntato da `c[ 6 ]` e, in generale, l'*i*-esimo elemento del vettore `c` sarà puntato da `c[ i-1 ]`. I nomi dei vettori, come i nomi delle altre variabili,

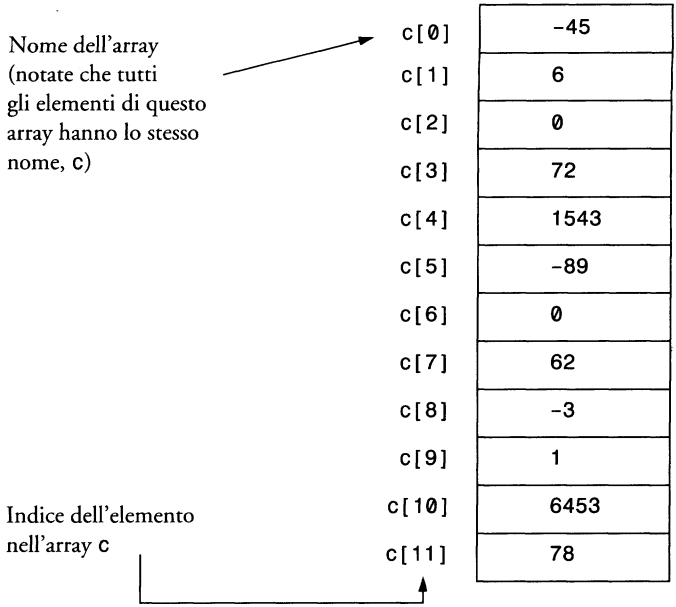


Figura 6.1 Un vettore di 12 elementi

possono contenere soltanto lettere, cifre e caratteri di sottolineatura. I nomi dei vettori non possono iniziare con una cifra.

Il numero di posizione contenuto all'interno delle parentesi quadre è più formalmente noto come *indice*. Un indice deve essere un intero o un'espressione intera. Nel caso che un programma utilizzi come indice un'espressione, questa sarà valutata per determinare l'indice. Per esempio, se `a = 5` e `b = 6`, allora l'istruzione

```
c[ a + b ] += 2;
```

aggiungerà 2 all'elemento `c[ 11 ]` del vettore. Osservate che un nome di vettore indicizzato è un lvalue: ovverosia un'entità che può essere utilizzata nella parte sinistra di un assegnamento.

Esaminiamo più da vicino il vettore `c` della Figura 6.1. Il *nome* del vettore è `c`. I suoi dodici elementi sono puntati da `c[ 0 ]`, `c[ 1 ]`, `c[ 2 ]`, ... `c[ 11 ]`. Il *valore* di `c[ 0 ]` è `-45`, quello di `c[ 1 ]` è `6`, quello di `c[ 2 ]` è `0`, quello di `c[ 7 ]` è `62` e quello di `c[ 11 ]` è `78`. Per visualizzare la somma dei valori contenuti nei primi tre elementi del vettore `c`, scriveremmo

```
printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );
```

Se volessimo dividere per 2 il valore del settimo elemento del vettore `c` e assegnare il risultato alla variabile `x`, scriveremmo

```
x = c[ 6 ] / 2;
```

Errore tipico 6.1

È importante notare la differenza tra "il settimo elemento del vettore" e "l'elemento sette del vettore". Dato che gli indici dei vettori incominciano da zero, "il settimo elemento del vettore" avrà l'indice 6, mentre "l'elemento sette del vettore" avrà l'indice 7 e sarà in realtà l'ottavo elemento del vettore. Questa è la causa di errori di " imprecisione di uno".

In C le parentesi quadre utilizzate per racchiudere l'indice di un vettore sono considerate in realtà un operatore. Esse hanno lo stesso livello di priorità dell'operatore di chiamata di funzione (cioè, le parentesi che vengono messe dopo un nome di funzione per invocarla). La Figura 6.2 mostra la priorità e l'associatività degli operatori introdotti sino a questo punto nel libro. Essi sono mostrati dall'alto in basso in ordine decrescente di priorità.

Operatori	Associatività	Tipo
`( ) [ ]`	da sinistra a destra	massima
`++ -- ! (`*tipo*`)`	da destra a sinistra	unari
`* / %`	da sinistra a destra	moltiplicativi
`+ -`	da sinistra a destra	additivi
`< <= > >=`	da sinistra a destra	relazionali
`== !=`	da sinistra a destra	di uguaglianza
`&&`	da sinistra a destra	AND logico
`||`	da sinistra a destra	OR logico
`?:`	da destra a sinistra	condizionale
`= += -= *= /= %=`	da destra a sinistra	di assegnamento
`,`	da sinistra a destra	virgola

Figura 6.2 Priorità degli operatori

6.3 La dichiarazione dei vettori

I vettori occupano dello spazio in memoria. Il programmatore specificherà il tipo di ogni elemento e il numero di quelli richiesti da ognuno dei vettori, così che il computer possa riservare l'appropriata quantità di memoria. Per ordinare al computer di riservare 12 elementi per il vettore di interi `c` sarà utilizzata la dichiarazione:

```
int c[ 12 ];
```

La seguente dichiarazione:

```
int b[ 100 ], x[ 27 ];
```

riserverà 100 elementi per il vettore di interi `b` e 27 per il vettore di interi `x`.

I vettori potranno essere dichiarati anche per contenere altri tipi di dato. Per esempio, un vettore di tipo `char` potrà essere utilizzato per immagazzinare una stringa di caratteri. Le stringhe di caratteri e la loro somiglianza con i vettori saranno discusse nel Capitolo 8. La relazione tra i puntatori e i vettori sarà invece discussa nel Capitolo 7.

6.4 Esempi sui vettori

Questa sezione presenta numerosi esempi che dimostrano come definire dei vettori, come inizializzarli e come compiere molte manipolazioni comuni su di essi.

Definire un vettore e utilizzare un ciclo per inizializzarne gli elementi

Il programma della Figura 6.3 utilizzerà dei comandi `for`, per azzerare i dieci elementi interi di un vettore `n` e visualizzarlo in formato tabulare. La prima istruzione `printf` (riga 16) visualizza le intestazioni per le due colonne stampate nel successivo comando `for`.

```
1  /* Fig. 6.3: fig06_03.c
2     inizializzare un vettore */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int n[ 10 ]; /* n è un vettore di 10 interi */
9     int i; /* contatore */
10
11    /* inizializza a 0 gli elementi del vettore n */
12    for ( i = 0; i < 10; i++ ) {
13       n[ i ] = 0; /* imposta l'elemento in posizione i a 0 */
14    } /* fine del comando for */
15
16    printf( "%s%13s\n", "Element", "Value" );
17
18    /* visualizza il contenuto del vettore n in formato tabulare */
19    for( i = 0; i < 10; i++ ) {
20       printf( "%7d%13d\n", i, n[ i ] );
21    } /* fine del comando for */
22
23    return 0; /* indica che il programma è terminato con successo */
24
25 } /* fine della funzione main */
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Figura 6.3 Inizializzare a zero gli elementi di un vettore

Inizializzare un vettore in una dichiarazione per mezzo di una lista di inizializzazione

Gli elementi di un vettore potranno anche essere inizializzati nel momento in cui il vettore viene dichiarato, facendo seguire alla dichiarazione un segno di uguale e delle parentesi graffe, {}, contenenti una lista di *inizializzatori* separati da virgole. Il programma della Figura 6.4 inizializza un vettore di interi con dieci valori (riga 9) e lo visualizza in formato tabulare.

Nel caso ci fossero meno inizializzatori degli elementi del vettore, quelli rimanenti sarebbero azzerati. Per esempio, gli elementi del vettore n nella Figura 6.3 potrebbero essere azzerati come segue:

```
int n[ 10 ] = { 0 };
```

In tal modo si azzererebbe il primo elemento in modo esplicito e in automatico i rimanenti nove, giacché ci sono meno inizializzatori degli elementi presenti nel vettore. È importante ricordare che i vettori non saranno azzerati automaticamente. Il programmatore dovrà azzerare almeno il primo elemento perché quelli rimanenti siano azzerati automaticamente. Questo metodo di azzeramento degli elementi di un vettore sarà eseguito durante la compilazione per i vettori dichiarati come `static` e a tempo di esecuzione per i vettori automatici.

```
1  /* Fig. 6.4: fig06_04.c
2     Inizializzare un vettore con una lista di inizializzazione */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     /* usa una lista di inizializzazione per inizializzare
         il vettore n */
9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    int i; /* contatore */
11
12    printf( "%s%13s\n", "Element", "Value" );
13
14    /* visualizza i contenuti del vettore in formato tabulare */
15    for( i = 0; i < 10; i++ ) {
16       printf( "%7d%13d\n", i, n[ i ] );
17    } /* fine del comando for */
18
19    return 0; /* indica che il programma è terminato con successo */
20
21 } /* fine della funzione main */
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
```

Figura 6.4 Inizializzare gli elementi di un vettore con una lista di inizializzazione (continua)

```
4     95
5     14
6     90
7     70
8     60
9     37
```

Figura 6.4 Inizializzare gli elementi di un vettore con una lista di inizializzazione

Errore tipico 6.2

Dimenticare di inizializzare gli elementi di un vettore qualora debbano essere inizializzati.

La dichiarazione di vettore

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

provocherà un errore di sintassi perché ci sono 6 inizializzatori e solo 5 elementi nel vettore.

Errore tipico 6.3

È un errore di sintassi fornire in una lista di inizializzazione di vettore più inizializzatori di quanti elementi ce ne siano nello stesso.

Qualora in una dichiarazione con una lista di inizializzazione sia stata omessa la dimensione del vettore, il numero dei suoi elementi sarà determinato da quello degli inizializzatori inclusi nella lista di inizializzazione. Per esempio,

```
int n[] = { 1, 2, 3, 4, 5 };
```

creerà un vettore di cinque elementi.

Specificare la dimensione di un vettore con una costante simbolica e inizializzare gli elementi del vettore con dei calcoli

Il programma della Figura 6.5 inizializzerà i 10 elementi di un vettore `s` con i valori `2`, `4`, `6`, ..., `20` e lo visualizzerà in formato tabulare. I valori saranno generati moltiplicando per `2` il contatore del ciclo e aggiungendovi `2`.

In questo programma è stata introdotta la direttiva del preprocessore `#define`. La riga 4

```
#define SIZE 10
```

definirà una *costante simbolica* `SIZE` il cui valore sarà `10`. Una costante simbolica è un identificatore che sarà sostituito con il *testo di sostituzione* dal preprocessore del C, prima che il programma sia compilato. Quando il programma sarà elaborato dal preprocessore, tutte le occorrenze della costante simbolica `SIZE` saranno sostituite dal testo di sostituzione `10`. Utilizzare le costanti simboliche per specificare le dimensioni dei vettori renderà i programmi più *scalabili*. Nella Figura 6.5, il primo ciclo `for` (riga 13) potrebbe riempire un vettore di 1000 elementi cambiando semplicemente il valore di `SIZE` nella direttiva `#define` da `10` a `1000`. Qualora non fosse stata utilizzata la costante simbolica `SIZE`, avremmo dovuto cambiare il programma in tre posti distinti per adattarlo alla gestione di un vettore di 1000 elementi. Questa tecnica diventerà sempre più utile per scrivere dei programmi chiari, man mano che questi diventeranno più corposi.

Errore tipico 6.4

Terminare con un punto e virgola una direttiva `#define` *o* `#include` *del preprocessore. Ricordate che le direttive del preprocessore non sono istruzioni del linguaggio C.*

```
1   /* Fig. 6.5: fig06_05.c
2      Inizializzare gli elementi del vettore s con gli interi pari
       da 2 a 20 */
3   #include <stdio.h>
4   #define SIZE 10
5
6   /* l'esecuzione del programma inizia dalla funzione main */
7   int main()
8   {
9      /* la costante simbolica SIZE può essere usata per specificare
          la dimensione del vettore */
10     int s[ SIZE ]; /* il vettore s ha 10 elementi */
11     int j; /* contatore */
12
13     for ( j = 0; j < SIZE; j++ ) {   /* imposta i valori */
14        s[ j ] = 2 + 2 * j;
15     } /* fine del comando for */
16
17     printf( "%s%13s\n", "Element", "Value" );
18
19     /* visualizza i contenuti del vettore s in formato tabulare */
20     for ( j = 0; j < SIZE; j++ ) {
21        printf( "%7d%13d\n", j, s[ j ] );
22     } /* fine del comando for */
23
24     return 0; /* indica che il programma è terminato con successo */
25
26  } /* fine della funzione main */
```

```
Element         Value
      0             2
      1             4
      2             6
      3             8
      4            10
      5            12
      6            14
      7            16
      8            18
      9            20
```

Figura 6.5 Generare i valori da inserire negli elementi di un vettore

Se la precedente direttiva del preprocessore nella riga 4 fosse stata terminata con un punto e virgola, tutte le occorrenze della costante simbolica `SIZE` incluse nel programma sarebbero state sostituite dal preprocessore con il testo `10;`. Ciò avrebbe potuto provocare degli errori di sintassi durante la compilazione o degli errori logici durante l'esecuzione. Ricordate che il preprocessore non è il C: è solo un manipolatore di testo.

Errore tipico 6.5

Assegnare un valore a una costante simbolica in una istruzione eseguibile è un errore di sintassi. Una costante simbolica non è una variabile. Il compilatore non le riserva nessuno spazio di memoria, come accade invece con le variabili che contengono dei valori durante l'esecuzione.

Ingegneria del software 6.1

Definire la dimensione di ogni vettore con una costante simbolica renderà i programmi più scalabili.

Buona abitudine 6.1

Utilizzate soltanto delle lettere maiuscole per i nomi delle costanti simboliche. Ciò le evidenzierà all'interno del programma e ricorderà al programmatore che le costanti simboliche non sono delle variabili.

Buona abitudine 6.2

Nei nomi delle costanti simboliche composti da più parole utilizzate i caratteri di sottolineatura per separare queste ultime in modo da aumentare la leggibilità.

Sommare gli elementi di un vettore

Il programma nella Figura 6.6 sommerà i valori contenuti in un vettore a di 12 elementi interi. Il corpo (riga 16) del comando `for` eseguirà il calcolo del totale.

```
1   /* Fig. 6.6: fig06_06.c
2      Calcolare la somma degli elementi del vettore */
3   #include <stdio.h>
4   #define SIZE 12
5
6   /* l'esecuzione del programma inizia dalla funzione main */
7   int main()
8   {
9      /* usa una lista di inizializzazione per inizializzare
          il vettore */
10     int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11     int i; /* contatore */
12     int total = 0; /* somma del vettore */
13
14     /* somma i contenuti del vettore a */
15     for ( i = 0; i < SIZE; i++ ) {
```

Figura 6.6 Calcolare la somma degli elementi di un vettore (continua)

```
16          total += a[ i ];
17       } /* fine del comando for */
18
19       printf( "Total of array element values is %d\n", total );
20
21       return 0; /* indica che il programma è terminato con successo */
22
23    } /* fine della funzione main */
```

```
Total of array element values is 383
```

Figura 6.6 Calcolare la somma degli elementi di un vettore

Usare i vettori per riassumere i risultati delle indagini

Il nostro prossimo esempio utilizzerà dei vettori per riassumere i risultati dei dati raccolti durante un'indagine. Considerate l'enunciato del problema.

È stato chiesto a quaranta studenti di fornire una valutazione sulla qualità del cibo servito nella mensa dello studente, in conformità a una scala da 1 a 10 (1 significa terribile e 10 eccellente). Sistemate i quaranta responsi in un vettore di interi e riassumete i risultati del sondaggio.

Questa è una tipica applicazione dei vettori (consultate la Figura 6.7). Vogliamo riassumere il numero dei responsi di ogni tipo (vale a dire, da 1 a 10). Il vettore `responses` (riga 17) sarà dunque formato da 40 elementi che conterranno le risposte degli studenti. Utilizzeremo un vettore di 11 elementi, `frequency` (riga 14), per contare il numero di occorrenze di ogni responso. Ignoreremo `frequency[0]` perché è logico incrementare di 1 il responso `frequency[1]` piuttosto che `frequency[0]`. Ciò ci consentirà di utilizzare direttamente ogni responso come indice del vettore `frequency`.

Buona abitudine 6.2

Adoperatevi per la chiarezza del programma. A volte potrà essere utile rinunciare a un uso più efficiente della memoria, o del tempo del processore, in favore di una scrittura più chiara dei programmi.

Obiettivo efficienza 6.1

A volte l'elemento dell'efficienza ha molta più importanza di quello della chiarezza.

Il primo ciclo `for` (riga 24) rileverà le risposte, una per volta, dal vettore `responses` e incrementerà uno dei 10 contatori (da `frequency[1]` a `frequency[10]`) inclusi nel vettore `frequency`. L'istruzione principale nel ciclo è la riga 25:

```
++frequency[ responses[ answer ] ];
```

che incrementerà in `frequency` il contatore appropriato secondo il valore di `responses[ answer ]`. Quando la variabile contatore `answer` varrà `0`, `responses[ answer ]` corrisponderà a `responses[ 0 ]` che varrà 1, perciò `++frequency[ responses[ answer ] ];` sarà effettivamente interpretata in realtà come

```
++frequency[ 1 ];
```

```
1  /* Fig. 6.7: fig06_07.c
2     Programma per il sondaggio degli studenti */
3  #include <stdio.h>
4  #define RESPONSE_SIZE 40 /* definisce le dimensioni dei vettori */
5  #define FREQUENCY_SIZE 11
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    int answer; /* contatore per iterare sulle 40 risposte */
11    int rating; /* contatore per iterare sulle frequenze 1-10 */
12
13    /* inizializza i contatori delle frequenze a 0 */
14    int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16    /* immette le risposte all'indagine nel vettore responses */
17    int responses[ RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18       1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19       5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    /* per ogni risposta, seleziona il valore di un elemento
         del vettore responses
22       e usa quel valore come indice nel vettore frequency
23       per determinare l'elemento da incrementare */
24    for( answer = 0; answer < RESPONSE_SIZE; answer++ ) {
25       ++frequency[ responses[ answer ] ];
26    } /* fine del comando for */
27
28    /* visualizza i  risultati */
29    printf( "%s%17s\n", "Rating", "Frequency" );
30
31    /* visualizza le frequenze in formato tabulare */
32    for( rating = 1; rating < FREQUENCY_SIZE; rating++ ) {
33       printf( "%6d%17d\n", rating, frequency[ rating ] );
34    } /* fine del comando for */
35
36    return 0; /* indica che il programma è terminato con successo */
37
38 } /* fine della funzione main */
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

Figura 6.7 Un programma per l'analisi del sondaggio degli studenti

che incrementerà l'elemento uno del vettore. Quando answer varrà 1, responses[answer] corrisponderà a responses[1] che varrà 2, perciò ++frequency[responses[answer]]; sarà effettivamente interpretata come

```
++frequency[ 2 ];
```

che incrementerà l'elemento due del vettore. Quando answer varrà 2, responses[answer] corrisponderà a responses[2] che varrà 6, perciò ++frequency[responses[answer]]; sarà effettivamente interpretata come

```
++frequency[ 6 ];
```

che incrementerà l'elemento sei del vettore e così via. Osservate che, indipendentemente dal numero delle risposte elaborate nel sondaggio, per riassumere i risultati sarà necessario un vettore di solo 11 elementi (ignorando quello con indice 0). Qualora i dati contenessero dei valori non validi, come 13, il programma avrebbe tentato di aggiungere 1 a frequency[13]. Questo indice però avrebbe superato i limiti del vettore. *Il C non ha nessun controllo sui limiti dei vettori che consenta di evitare che il computer faccia riferimento a un elemento inesistente.* Ne consegue che un programma in esecuzione potrebbe "oltrepassare" i confini di un vettore senza alcun avviso. Il programmatore dovrà quindi assicurarsi che tutti i riferimenti al vettore rimangano all'interno dei suoi limiti.

Errore tipico 6.6

Fare riferimento a un elemento esterno ai limiti del vettore.

Collaudo e messa a punto 6.1

Quando scorrete un vettore, il suo indice non dovrebbe mai scendere sotto lo 0 e dovrebbe essere sempre inferiore al numero totale degli elementi inclusi nel vettore (dimensione – 1). Assicuratevi che la condizione di terminazione del ciclo prevenga l'accesso a elementi esterni al suddetto intervallo.

Collaudo e messa a punto 6.2

I programmi dovrebbero verificare la correttezza di tutti i valori presi in input, in modo da evitare che delle informazioni errate possano influenzare i calcoli di un programma.

Rappresentare graficamente i valori degli elementi di un vettore con gli istogrammi

Il nostro prossimo esempio (Figura 6.8) leggerà alcuni numeri da un vettore e presenterà quelle informazioni in un grafico a barre o istogramma: a fianco a ogni numero sarà visualizzato una barra composta di altrettanti asterischi. Il comando for nidificato (riga 20) disegnerà le barre. Osservate l'utilizzo di printf("\n") per terminare ogni barra dell'istogramma (riga 24).

```
1  /* Fig. 6.8: fig06_08.c
2     Programma per la visualizzazione di istogrammi */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* l'esecuzione del programma inizia dalla funzione main */
```

Figura 6.8 Visualizzazione di istogrammi (continua)

```
 7  int main()
 8  {
 9     /* usa una lista di inizializzazione per inizializzare
          il vettore n */
10     int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
11     int i; /* contatore del for esterno per gli elementi
                 del vettore */
12     int j; /* contatore del for interno, conta gli asterischi in ogni
                barra dell'istogramma */
13
14     printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
15
16     /* visualizza una barra dell'istogramma per ogni elemento
          del vettore n */
17     for ( i = 0; i < SIZE; i++ ) {
18        printf( "%7d%13d        ", i, n[ i ] );
19
20        for( j = 1; j <= n[ i ]; j++ ) {   /* visualizza una barra */
21           printf( "%c", '*' );
22        } /* fine del comando for interno */
23
24        printf( "\n" ); /* termina una barra dell'istogramma */
25     } /* fine del comando for esterno */
26
27     return 0; /* indica che il programma è terminato con successo */
28
29  } /* fine della funzione main */
```

```
Element         Value         Histogram
      0            19         ********************
      1             3         ***
      2            15         ***************
      3             7         *******
      4            11         ***********
      5             9         *********
      6            13         *************
      7             5         *****
      8            17         *****************
      9             1         *
```

Figura 6.8 Visualizzazione di istogrammi

Lanciare un dado 6000 volte e riassumere i risultati in un vettore

Nel Capitolo 5, abbiamo affermato che vi avremmo mostrato un metodo più elegante di quello della Figura 5.8 per scrivere il programma di lancio dei dadi. Il problema richiedeva di lanciare 6000 volte un dado con sei facce, in modo da verificare se il generatore di numeri casuali producesse effettivamente dei valori a caso. Nella Figura 6.9 è mostrata una versione con vettori dello stesso programma.

```
1  /* Fig. 6.9: fig06_09.c
2     Lanciare 6000 volte un dado a sei facce */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6  #define SIZE 7
7
8  /* l'esecuzione del programma inizia dalla funzione main */
9  int main()
10 {
11    int face;  /* valore casuale del dado 1-6 */
12    int roll; /* contatore dei lanci 1-6000 */
13    int frequency[ SIZE ] = { 0 }; /* azzera i conteggi */
14
15    srand( time( NULL ) ); /* seme per il generatore di numeri
                                casuali */
16
17    /* lancia il dado 6000 volte */
18    for ( roll = 1; roll <= 6000; roll++ ) {
19       face = rand() % 6 + 1;
20       ++frequency[ face ];       /* sostituisce lo switch di 26
                                      righe della Figura 5.8 */
21    } /* fine del comando for */
22
23    printf( "%s%17s\n", "Face", "Frequency" );
24
25    /* visualizza gli elementi 1-6 di frequency in formato tabulare */
26    for ( face = 1; face < SIZE; face++ ) {
27       printf( "%4d%17d\n", face, frequency[ face ] );
28    } /* fine del comando for */
29
30    return 0;  /* indica che il programma è terminato con successo */
31
32 } /* fine della funzione main */
```

```
Face        Frequency
   1             1029
   2              951
   3              987
   4             1033
   5             1010
   6              990
```

Figura 6.9 Programma per il lancio di un dado che utilizza i vettori invece di uno switch

Usare vettori di caratteri per memorizzare e manipolare le stringhe

Abbiamo discusso soltanto dei vettori di interi. I vettori però sono in grado di conservare dati di qualsiasi tipo. Discuteremo ora dell'immagazzinamento delle stringhe nei vettori di caratteri. Fino a questo punto, l'unica capacità di elaborazione delle stringhe di cui abbiamo

trattato è stata la stampa di una stringa con `printf`. Nel linguaggio C una stringa come "`hello`" è in realtà un vettore di caratteri individuali.

I vettori di caratteri hanno molte qualità singolari. Essi possono essere inizializzati utilizzando una stringa letterale. Per esempio la dichiarazione

```
char string1[] = "first";
```

inizializzerà gli elementi del vettore `string1` con i caratteri individuali della stringa letterale "`first`". La dimensione del vettore `string1` in questo caso sarà determinata dal compilatore in base alla lunghezza della stringa. È importante notare che la stringa "`first`" conterrà cinque caratteri *più* uno speciale, detto *carattere nullo*, per la terminazione della stringa. Di conseguenza il vettore `string1` conterrà in realtà sei elementi. La rappresentazione come costante di carattere del carattere nullo è '`\0`'. Tutte le stringhe in C terminano con questo carattere. Un vettore di caratteri che rappresenti una stringa dovrà sempre essere dichiarato con una dimensione sufficiente a contenere il numero dei caratteri della stringa e quello nullo di terminazione.

I vettori di caratteri possono anche essere inizializzati con singole costanti di carattere sistemate all'interno di una lista di inizializzatori. La dichiarazione precedente è equivalente a

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Dato che una stringa è in realtà un vettore di caratteri, potremo accedere direttamente a ognuno di essi utilizzando la notazione con gli indici di vettore. Per esempio, `string1[ 0 ]` sarà il carattere '`f`' mentre `string1[ 3 ]` sarà '`s`'.

Potremo anche prendere in input dalla tastiera una stringa e sistemarla direttamente in un vettore di caratteri, utilizzando `scanf` e la specifica di conversione `%s`. Per esempio, la dichiarazione

```
char string2[ 20 ];
```

creerà un vettore di caratteri in grado di immagazzinare una stringa di 19 caratteri oltre a quello nullo di terminazione. L'istruzione

```
scanf( "%s", string2 );
```

leggerà una stringa dalla tastiera e la immagazzinerà in `string2`. Osservate che il nome del vettore è stato passato alla `scanf` senza farlo precedere da `&`, come abbiamo fatto con le altre variabili. Normalmente l'operatore `&` è utilizzato per indicare a `scanf` la posizione di una variabile all'interno della memoria, così che vi ci possa immagazzinare un valore. Nella sezione 6.5 discuteremo del passaggio dei vettori alle funzioni. Vedremo che il nome di un vettore è in realtà l'indirizzo del suo primo elemento; è proprio per questo motivo che l'operatore `&` non è necessario con la `scanf`.

È responsabilità del programmatore assicurarsi che il vettore destinatario sia in grado di contenere qualsiasi stringa l'utente possa digitare alla tastiera. La funzione `scanf` leggerà i dati dalla tastiera finché non avrà incontrato il primo carattere di spazio bianco: non si preoccuperà di quanto sia grande il vettore destinatario. Di conseguenza, `scanf` potrà scrivere anche oltre la fine del vettore.

Errore tipico 6.7

Non fornire a `scanf` *un vettore di caratteri sufficientemente grande per immagazzinare una stringa digitata alla tastiera, potrà causare in un programma una distruzione di dati e altri errori durante l'esecuzione.*

Un vettore di caratteri che rappresenti una stringa potrà essere inviato in output con `printf` e la specifica di conversione `%s`. Il vettore `string2` sarà visualizzato con l'istruzione

```
printf( "%s\n", string2 );
```

Osservate che anche `printf`, come `scanf`, non si preoccupa di quanto sia grande il vettore di caratteri. I caratteri della stringa saranno visualizzati finché non verrà incontrato un carattere nullo di terminazione.

La Figura 6.10 mostra l'inizializzazione di un vettore di caratteri con una stringa letterale, la lettura di una stringa in un vettore di caratteri, la stampa di un vettore di caratteri come una stringa e l'accesso ai singoli caratteri di una stringa.

```
1  /* Fig. 6.10: fig06_10.c
2     Trattare i vettori di caratteri come stringhe */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     char string1[ 20 ]; /* riserva lo spazio per 20 caratteri */
9     char string2[] = "string literal"; /* riserva lo spazio per 15
                                            caratteri */
10    int i; /* contatore */
11
12    /* legge la stringa proveniente dall'utente nel vettore string1 */
13    printf( "Enter a string: " );
14    scanf( "%s", string1 );
15
16    /* visualizza le stringhe */
17    printf( "string1 is: %s\nstring2: is %s\n"
18           "string1 with spaces between characters is:\n",
19           string1, string2 );
20
21    /* visualizza i caratteri fino a raggiungere il carattere nullo */
22    for ( i = 0; string1[ i ] != '\0'; i++ ) {
23       printf( "%c ", string1[ i ] );
24    } /* fine del comando for */
25
26    printf( "\n" );
27
28    return 0; /* indica che il programma è terminato con successo */
29
30 } /* fine della funzione main */
```

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Figura 6.10 Trattare i vettori di caratteri come stringhe

La Figura 6.10 utilizza un comando `for` (riga 22) per scorrere il vettore `string1` e visualizzare i singoli caratteri, separati da spazi, utilizzando la specifica di conversione `%c`. La condizione del comando `for`, `string1[ i ] != '\0'`, sarà vera finché nella stringa non verrà incontrato il carattere nullo di terminazione.

Vettori locali statici e vettori locali automatici

Il Capitolo 5 ha trattato la specifica di classe di memoria `static`. Un variabile locale `static` esisterà per tutta la durata di un programma, ma sarà visibile soltanto nel corpo della funzione. Potremo applicare `static` alla dichiarazione di un vettore locale, per fare in modo che questo non sia creato e inizializzato ogni volta che la funzione sarà invocata, e che non sia distrutto ogni volta che il programma uscirà dalla funzione. Ciò ridurrà il tempo di esecuzione del programma, in modo particolare per quelli che prevedono frequenti invocazioni di funzioni che dichiarano dei vettori di dimensioni ragguardevoli.

Obiettivo efficienza 6.2

Qualora una funzione cambiasse frequentemente il suo stato di visibilità e contenesse dei vettori automatici, questi dovrebbero essere resi `static` *in modo da evitare che siano creati ogni volta che la funzione sarà richiamata.*

I vettori dichiarati `static` saranno inizializzati automaticamente una sola volta, durante la fase di compilazione. Qualora un vettore `static` non fosse stato inizializzato esplicitamente dal programmatore, sarebbe azzerato dal compilatore.

La Figura 6.11 mostra la funzione `staticArrayInit` (riga 24) con un vettore locale `static` (riga 27), e una funzione `automaticArrayInit` (riga 47) con un vettore locale automatico (riga 50). La funzione `staticArrayInit` sarà richiamata due volte (righe 12 e 16). Il vettore locale `static` dichiarato nella funzione sarà azzerato dal compilatore (riga 27). La funzione visualizzerà il vettore, aggiungerà 5 a ogni suo elemento e lo visualizzerà nuovamente. Quando la funzione verrà richiamata per la seconda volta, il vettore `static` conterrà i valori immagazzinati durante la prima invocazione della funzione. Anche la funzione `automaticArrayInit` sarà richiamata due volte (righe 13 e 17). Gli elementi del vettore locale automatico dichiarato nella funzione saranno inizializzati con i valori 1, 2 e 3 (riga 50). La funzione visualizzerà il vettore, aggiungerà 5 a ogni suo elemento e lo visualizzerà nuovamente. Quando la funzione verrà richiamata per la seconda volta, gli elementi del vettore saranno nuovamente inizializzati a 1, 2 e 3 poiché il vettore avrà una permanenza automatica nella memoria.

Errore tipico 6.8

Presumere che gli elementi di un vettore locale `static` *siano azzerati ogni volta che sarà richiamata la funzione in cui il vettore è stato dichiarato.*

```
1  /* Fig. 6.11: fig06_11.c
2     I vettori statici sono inizializzati a zero */
3  #include <stdio.h>
4
5  void staticArrayInit( void ); /* prototipo di funzione */
6  void automaticArrayInit( void ); /* prototipo di funzione */
```

Figura 6.11 I vettori statici saranno azzerati automaticamente, qualora non siano stati inizializzati esplicitamente dal programmatore (continua)

```
 7
 8  /* l'esecuzione del programma inizia dalla funzione main */
 9  int main()
10  {
11     printf( "First call to each function:\n" );
12     staticArrayInit();
13     automaticArrayInit();
14
15     printf( "\n\nSecond call to each function:\n" );
16     staticArrayInit();
17     automaticArrayInit();
18
19     return 0; /* indica che il programma è terminato con successo */
20
21  } /* fine della funzione main */
22
23  /* funzione per dimostrare l'uso di un vettore locale statico */
24  void staticArrayInit( void )
25  {
26     /* la prima volta che la funzione viene chiamata azzera gli elementi */
27     static int a[ 3 ];
28     int i; /* contatore */
29
30     printf( "\nValues on entering staticArrayInit:\n" );
31
32     /* visualizza i contenuti di array1 */
33     for ( i = 0; i <= 2; i++ ) {
34        printf( "array1[ %d ] = %d  ", i, a[ i ] );
35     } /* fine del comando for */
36
37     printf( "\nValues on exiting staticArrayInit:\n" );
38
39     /* modifica e visualizza i contenuti di array1 */
40     for ( i = 0; i <= 2; i++ ) {
41        printf( "array1[ %d ] = %d  ", i, a[ i ] += 5 );
42     } /* fine del comando for */
43
44  } /* fine della funzione staticArrayInit */
45
46  /* funzione per mostrare l'uso di un vettore locale automatico */
47  void automaticArrayInit( void )
48  {
49     /* inizializza gli elementi ogni volta che la funzione viene
           chiamata */
50     int a[ 3 ] = { 1, 2, 3 };
51     int i; /* contatore */
52
53     printf( "\n\nValues on entering automaticArrayInit:\n" );
54
```

Figura 6.11 I vettori statici saranno azzerati automaticamente, qualora non siano stati inizializzati esplicitamente dal programmatore (continua)

```
55       /* visualizza i contenuti di array2 */
56       for ( i = 0; i <= 2; i++ ) {
57          printf( "array2[ %d ] = %d  ", i, a[ i ] );
58       } /* fine del comando for */
59
60       printf( "\nValues on exiting automaticArrayInit:\n" );
61
62       /* modifica e visualizza i contenuti di array2 */
63       for ( i = 0; i <= 2; i++ ) {
64          printf( "array2[ %d ] = %d  ", i, a[ i ] += 5 );
65       } /* fine del comando for */
66
67    } /* fine della funzione automaticArrayInit */
```

```
First call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 0  array1[ 1 ] = 0  array1[ 2 ] = 0
Values on exiting staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5  array1[ 1 ] = 5  array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10  array1[ 1 ] = 10  array1[ 2 ] = 10

Values on entering automaticArrayInit:
array2[ 0 ] = 1  array2[ 1 ] = 2  array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6  array2[ 1 ] = 7  array2[ 2 ] = 8
```

Figura 6.11 I vettori statici saranno azzerati automaticamente, qualora non siano stati inizializzati esplicitamente dal programmatore

6.5 Passare i vettori alle funzioni

Indicherete il nome del vettore senza parentesi quadre per passare a una funzione un argomento di quel tipo. Per esempio, se il vettore `hourlyTemperatures` fosse stato dichiarato come

```
int hourlyTemperatures[24];
```

l'istruzione per la chiamata della funzione

```
modifyArray(hourlyTemperatures, 24);
```

passerebbe il vettore `hourlyTemperatures` e la sua dimensione alla funzione `modifyArray`. A differenza dei vettori che contengono stringhe, gli altri tipi di vettori non hanno un terminatore speciale. Per questa ragione la dimensione di un vettore viene passata alla funzione in modo che quest'ultima possa elaborare il numero corretto di elementi.

Il C passa i vettori alle funzioni automaticamente per riferimento : la funzione chiamata potrà modificare i valori degli elementi inclusi nel vettore originale del chiamante. Il nome del vettore è in realtà l'indirizzo del suo primo elemento. Dato che sarà passato l'indirizzo iniziale del vettore, la funzione chiamata conoscerà precisamente le locazioni in cui quest'ultimo è stato memorizzato. Di conseguenza, quando all'interno del suo corpo la funzione chiamata modificherà gli elementi del vettore, essa starà modificando effettivamente quelli del chiamante direttamente nelle loro locazioni di memoria originarie.

La Figura 6.12 dimostrerà che il nome di un vettore è in realtà l'indirizzo del suo primo elemento, visualizzando `array`, `&array[ 0 ]` e `&array` usando `%p`: una speciale specifica di conversione usata per visualizzare gli indirizzi. La specifica di conversione `%p` normalmente visualizza gli indirizzi come numeri esadecimali. I numeri esadecimali (in base 16) sono formati dalle cifre da 0 a 9 e dalle lettere da A a F (tali lettere sono gli equivalenti esadecimali dei numeri 10-15). Essi sono utilizzati spesso come notazione abbreviata per dei valori interi grandi. L'Appendice E, "I sistemi numerici", fornisce una trattazione approfondita sulle relazioni tra gli interi binari (in base 2), gli ottali (in base 8), i decimali (in base 10; gli interi standard) e gli esadecimali. L'output mostrerà che `array` e `&array[ 0 ]` avranno lo stesso valore, vale a dire `0012FF78`. L'output di questo programma dipenderà dal sistema, ma gli indirizzi saranno sempre identici per una data esecuzione di questo programma su un dato computer.

Obiettivo efficienza 6.3

Il passaggio dei vettori per riferimento ha un senso per ragioni di efficienza. Infatti, passare un vettore per valore significa passare una copia di ognuno dei suoi elementi. Ovviamente, per dei vettori di dimensioni ragguardevoli passati frequentemente, ciò sarebbe dispendioso in termini di tempo e si consumerebbe una considerevole quantità di memoria per la loro copia.

```
1  /* Fig. 6.12: fig.06_12.c
2     Il nome di un vettore è equivalente a &array[0] */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     char array[ 5 ]; /* definisce un vettore di dimensione 5 */
9
10    printf( "    array = %p\n&array[ 0 ] = %p\n"
11            "    &array = %p\n",
12            array, &array[ 0 ], &array );
13
14    return 0; /* indica che il programma è terminato con successo */
15
16 } /* fine della funzione main */
```

Figura 6.12 Il nome di un vettore è equivalente all'indirizzo del suo primo elemento (continua)

```
      array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

Figura 6.12 Il nome di un vettore è equivalente all'indirizzo del suo primo elemento

Ingegneria del software 6.2

Sarà possibile passare un vettore per valore (utilizzando un semplice trucco che spiegheremo nel Capitolo 10).

Se da un lato i vettori completi sono passati per riferimento, dall'altro i suoi singoli elementi sono passati per valore esattamente come accade per le semplici variabili. Questi semplici singoli pezzi di dato (ad esempio singoli elementi di tipo `int`, `float` e `char`) sono detti *scalari*. Per passare a una funzione un elemento di un vettore, utilizzerete il suo nome indicizzato come argomento nella chiamata della funzione. Nel Capitolo 7, mostreremo come passare per riferimento alle funzioni gli scalari (ovverosia, le singole variabili e gli elementi dei vettori).

La lista dei parametri di una funzione dovrà specificare esplicitamente che sarà ricevuto un vettore, affinché questo possa essere ricevuto attraverso la chiamata. Per esempio, l'intestazione per la funzione `modifyArray` (che abbiamo richiamato precedentemente in questa sezione) potrebbe essere scritta come

```
void modifyArray( int b[], int size )
```

e indicherebbe che `modifyArray` accetterà nel parametro `b` un vettore di interi e in `size` il numero degli elementi inclusi nello stesso. La dimensione del vettore all'interno delle parentesi quadre non è obbligatoria. Qualora fosse stata inclusa, il compilatore avrebbe controllato che fosse maggiore di zero e l'avrebbe ignorata. Specificare una dimensione negativa è un errore di compilazione. Dato che i vettori sono passati automaticamente per riferimento, quando la funzione chiamata utilizzerà il nome del vettore `b`, farà riferimento a quello dichiarato dal chiamante (il vettore `hourlyTemperatures` nella chiamata precedente). Nel Capitolo 7 introdurremo altre notazioni per indicare che un vettore verrà ricevuto da una funzione. Queste notazioni, come vedremo, sono basate sulla stretta correlazione che intercorre tra i vettori e i puntatori del linguaggio C.

Il programma della Figura 6.13 mostrerà la differenza tra il passaggio di un intero vettore e quello di un suo elemento. Il programma visualizzerà in primo luogo i cinque elementi del vettore di interi `a` (righe 20-22). In seguito, `a` e la sua dimensione saranno passati alla funzione `modifyArray` (riga 27), dove ogni elemento di `a` sarà moltiplicato per 2 (righe 56-57). Quindi, `a` sarà visualizzato nuovamente all'interno di `main` (righe 32-34). Come dimostrerà l'output, gli elementi di `a` saranno stati indubbiamente modificati da `modifyArray`. A questo punto il programma visualizzerà il valore di `a[ 3 ]` (riga 38) e lo passerà alla funzione `modifyElement` (riga 40). Questa moltiplicherà per 2 (riga 67) i suoi argomenti e visualizzerà il loro nuovo valore. Osservate che, quando `a[ 3 ]` sarà visualizzato nuovamente all'interno di `main` (riga 43), esso non risulterà modificato perché gli elementi individuali dei vettori saranno stati passati per valore.

```
1  /* Fig.6.13: fig06_13.c
2     Passare alle funzioni i vettori e i singoli elementi
      di un vettore */
3  #include <stdio.h>
4  #define SIZE 5
5
6  /* prototipi di funzioni */
7  void modifyArray( int b[], int size );
8  void modifyElement( int e );
9
10 /* l'esecuzione del programma inizia dalla funzione main */
11 int main()
12 {
13    int a[ SIZE ] = { 0, 1, 2, 3, 4 }; /* inizializza a */
14    int i; /* contatore */
15
16    printf( "Effects of passing entire array by reference:\n\nThe "
17          "values of the original array are:\n" );
18
19    /* visualizza il vettore originale */
20    for ( i = 0; i < SIZE; i++ ) {
21       printf( "%3d", a[ i ] );
22    } /* fine del comando for */
23
24    printf( "\n" );
25
26    /* passa per riferimento il vettore a alla funzione modifyArray */
27    modifyArray( a, SIZE );
28
29    printf( "The values of the modified array are:\n" );
30
31    /* visualizza il vettore modificato */
32    for ( i = 0; i < SIZE; i++ ) {
33       printf( "%3d", a[ i ] );
34    } /* fine del comando for */
35
36    /* visualizza il valore di a[ 3 ] */
37    printf( "\n\n\nEffects of passing array element call "
38          "by value:\n\nThe value of a[ 3 ] is %d\n", a[ 3 ] );
39
40    modifyElement( a[ 3 ] ); /* passa per valore l'elemento
                                  del vettore a[ 3 ] */
41
42    /* visualizza il valore di a[ 3 ] */
43    printf( "The value of a[ 3 ] is %d\n", a[ 3 ]);
44
45    return 0; /* indica che il programma è terminato con successo */
46
```

Figura 6.13 Passare alle funzioni i vettori e i loro singoli elementi (continua)

```
47  } /* fine della funzione main */
48
49  /* nella funzione modifyArray, "b" punta al vettore originale "a"
50      in memoria */
51  void modifyArray( int b[], int size )
52  {
53     int j; /* contatore */
54
55     /* moltiplica ogni elemento del vettore per 2 */
56     for ( j = 0; j < size; j++ ) {
57        b[ j ] *= 2;
58     } /* fine del comando for */
59
60  } /* fine della funzione modifyArray */
61
62  /* nella funzione modifyElement, "e" è una copia locale dell'elemento
       del vettore
63     a[3] passato dal main */
64  void modifyElement( int e )
65  {
66     /* moltiplica il parametro per 2 */
67     printf( "Value in modifyElement is %d\n", e *= 2 );
68  } /* fine della funzione modifyElement */
```

```
Effects of passing entire array call by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element call by value:

The value of a[ 3 ] is 6
Value in modifyElement is 12
The value of a[ 3 ] is 6
```

Figura 6.13 Passare alle funzioni i vettori e i loro singoli elementi

Nei vostri programmi, ci saranno molte situazioni in cui a una funzione non dovrà essere consentito di modificare gli elementi di un vettore. Dato che i vettori saranno sempre passati per riferimento, sarà difficile controllare che i valori non siano modificati. Il C fornisce lo speciale qualificatore di tipo `const` proprio per prevenire, all'interno di una funzione, la modifica dei valori contenuti in un vettore. Quando un parametro di tipo vettore sarà preceduto dal qualificatore `const`, i suoi elementi diventeranno delle costanti nel corpo della funzione e ogni tentativo di modificarli, in quel contesto, provocherà un errore in fase di compilazione. Ciò consentirà al programmatore di correggere un programma in modo che non provi a modificare gli elementi del vettore.

La Figura 6.14 mostra l'uso del qualificatore `const`. La funzione `tryModifyArray` (riga 22) sarà definita con il parametro `const int b[]`, il quale dichiarerà il vettore `b` come una costante che, in quanto tale, non potrà essere modificata. L'output mostra i messaggi di errore che saranno emessi dal compilatore; tali messaggi potrebbero essere differenti nel vostro sistema. I tre tentativi della funzione di modificare gli elementi del vettore provocheranno ognuno l'errore "`l-value specifies a const object.`" ("Impossibile modificare un oggetto `const`.") del compilatore. Il qualificatore `const` sarà discusso nel Capitolo 7.

Ingegneria del software 6.3

Nella definizione di una funzione, il qualificatore di tipo `const` *potrà essere applicato a un parametro di tipo vettore, per evitare che il vettore originale sia modificato nel corpo della funzione. Questo è un altro esempio del principio del minimo privilegio. Alle funzioni non dovrebbe essere mai data la possibilità di modificare un vettore, sempre che non sia assolutamente necessario.*

```
1  /* Fig. 6.14: fig06_14.c
2     Dimostrare l'uso del qualificatore di tipo const con i vettori */
3  #include <stdio.h>
4
5  void tryToModifyArray( const int b[] ); /* prototipo di funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    int a[] = { 10, 20, 30 }; /* inizializza a */
11
12    tryToModifyArray( a );
13
14    printf( "%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15
16    return 0; /* indica che il programma è terminato con successo */
17
18 } /* fine della funzione main */
19
20 /* nella funzione tryToModifyArray, il vettore b è const, quindi non
21    può essere usato per modificare il vettore originale a nel main */
22 void tryToModifyArray( const int b[] )
23 {
24    b[ 0 ] /= 2;    /* errore */
25    b[ 1 ] /= 2;    /* errore */
26    b[ 2 ] /= 2;    /* errore */
27 } /* fine della funzione tryToModifyArray */
```

```
Compiling...
FIG06_14.C
fig06_14.c(24) : error C2166: l-value specifies const object
fig06_14.c(25) : error C2166: l-value specifies const object
fig06_14.c(26) : error C2166: l-value specifies const object
```

Figura 6.14 Il qualificatore di tipo const

6.6 L'ordinamento dei vettori

L'*ordinamento* dei dati (ovverosia, la loro sistemazione in un ordine particolare, come ascendente o discendente) è una delle applicazioni più importanti nel campo dell'elaborazione elettronica. Una banca ordina tutti gli assegni per numero di conto, così che possa preparare i singoli estratti conto alla fine di ogni mese. Le compagnie telefoniche ordinano i propri elenchi per cognome e, nell'ambito di quello, per nome, in modo da semplificare la ricerca dei numeri telefonici. In teoria, tutte le organizzazioni devono ordinare qualche dato e, in molti casi, delle massicce quantità di dati. L'ordinamento dei dati è un problema interessante che ha attirato alcuni degli sforzi più intensi della ricerca informatica. In questo capitolo, discuteremo quello che è forse lo schema di ordinamento più semplice tra quelli noti. Negli esercizi del Capitolo 12 esamineremo con cura degli schemi più complessi che forniscono prestazioni notevolmente superiori.

Obiettivo efficienza 6.4

Gli algoritmi più semplici sono spesso poco efficienti. La loro virtù è la facilità con cui possono essere scritti, collaudati e messi a punto. Ne consegue che saranno spesso necessari degli algoritmi più complessi per ottenere il massimo dell'efficienza.

Il programma nella Figura 6.15 ordinerà in modo ascendente i valori dei 10 elementi inclusi nel vettore a (riga 10). La tecnica che utilizzeremo è detta *bubble sort* (ordinamento a bolle) o *sinking sort* (ordinamento con sprofondamento), perché i valori più piccoli "risalgono" gradualmente a galla sino in cima al vettore (la prima posizione), proprio come nell'acqua le bolle d'aria risalgono in superficie, mentre i valori più grandi sprofondano verso il fondo del vettore (l'ultima posizione). La tecnica prevede l'esecuzione di diversi passaggi sul vettore. In ognuno di essi sarà confrontata ogni coppia di elementi adiacenti. Lasceremo i valori così come li avremo trovati, qualora una coppia sia già in ordine crescente (o nel caso che i valori siano identici). I loro valori saranno invece scambiati di posizione all'interno del vettore, qualora una coppia sia in ordine decrescente.

```
1   /* Fig. 6.14: fig06_14.c
2      Questo programma ordina in modo ascendente i valori di un vettore */
3   #include <stdio.h>
4   #define SIZE 10
5
6   /* l'esecuzione del programma inizia dalla funzione main */
7   int main()
8   {
9      /* inizializza a */
10     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11     int pass; /* contatore dei passaggi */
12     int i; /* contatore dei confronti */
13     int hold; /* locazione temporanea utilizzata per scambiare due
                    elementi del vettore */
14
15     printf( "Data items in original order\n" );
16
17     /* visualizza il vettore originale */
```

Figura 6.15 Ordinamento di un vettore con il bubble sort (continua)

```
18      for ( i = 0; i < SIZE; i++ ) {
19         printf( "%4d", a[ i ] );
20      } /* fine del comando for */
21
22      /* bubble sort */
23      /* ciclo per controllare il numero dei passaggi */
24      for ( pass = 1; pass < SIZE; pass++ ) {
25
26         /* ciclo per controllare il numero di confronti per ogni
              passaggio */
27         for ( i = 0; i < SIZE - 1; i++ ) {
28
29            /* confronta gli elementi adiacenti e li scambia se il primo
30               elemento è maggiore del secondo */
31            if ( a[ i ] > a[ i + 1 ] ) {
32               hold = a[ i ];
33               a[ i ] = a[ i + 1 ];
34               a[ i + 1 ] = hold;
35            } /* fine del comando if */
36
37         } /* fine del comando for interno */
38
39      } /* fine del comando for esterno */
40
41      printf( "\nData items in ascending order\n" );
42
43      /* visualizza il vettore ordinato */
44      for ( i = 0; i < SIZE; i++ ) {
45         printf( "%4d", a[ i ] );
46      } /* fine del comando for */
47
48      printf( "\n" );
49
50      return 0; /* indica che il programma è terminato con successo */
51   } /* fine della funzione main */

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Figura 6.15 Ordinamento di un vettore con il bubble sort

Il programma confronterà prima a[0] con a[1], quindi a[1] con a[2], poi a[2] con a[3] e così via fino al completamento del passaggio con il confronto tra a[8] e a[9]. Osservate che saranno eseguiti soltanto nove confronti sebbene ci siano 10 elementi. A causa del modo in cui saranno effettuati i confronti successivi, in un singolo passaggio, un valore grande potrà muoversi di molte posizioni verso il fondo di del vettore, mentre un valore piccolo potrà muoversi di una sola posizione verso l'alto. Dopo il primo passaggio, il

numero più grande si sarà sicuramente adagiato sul fondo del vettore, ovverosia `a[ 9 ]`. Dopo il secondo passaggio, il secondo valore più grande sarà sicuramente affondato in `a[ 8 ]`. Dopo il nono passaggio, il nono valore in ordine di grandezza sarà affondato in `a[ 1 ]`. Tutto ciò lascerà il valore più piccolo in `a[ 0 ]`, perciò saranno necessari solo nove passaggi sul vettore per ordinarlo, nonostante questo contenga dieci elementi.

L'ordinamento sarà eseguito dal ciclo `for` nidificato (righe 24-39). Qualora sia necessario uno scambio, questo sarà eseguito dai tre assegnamenti

```
hold = a[ i ];
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = hold;
```

dove la variabile supplementare `hold` immagazzinerà temporaneamente uno dei due valori che dovranno essere scambiati di posto. Lo scambio non può essere eseguito con due soli assegnamenti:

```
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = a[ i ];
```

Per esempio, se `a[ i ]` fosse 7 e `a[ i + 1 ]` fosse 5, dopo il primo assegnamento entrambi i valori sarebbero 5 e il 7 andrebbe perso, da cui la necessità della variabile supplementare `hold`.

La virtù principale dell'ordinamento a bolle è la sua facilità di implementazione. L'ordinamento a bolle è però lento. Ciò diventerà evidente quando si ordineranno dei vettori di dimensioni ragguardevoli. Negli esercizi svilupperemo delle versioni più efficienti dell'ordinamento a bolle. Sono anche stati sviluppati degli ordinamenti notevolmente più efficienti di quello a bolle. Negli esercizi esamineremo con cura alcuni di essi. I corsi più avanzati esamineranno più approfonditamente l'ordinamento e la ricerca dei dati.

6.7 Studio di un caso: calcolare la media, la mediana e la moda usando i vettori

Considereremo ora un esempio più corposo. I computer sono utilizzati comunemente per *l'analisi dei dati delle indagini* per compilare e analizzare i risultati delle indagini e dei sondaggi di opinione. Il programma della Figura 6.16 utilizzerà il vettore `response` inizializzato con 99 opinioni raccolte durante un'indagine. Ognuna delle opinioni sarà un numero da 1 a 9. Il programma calcolerà la media, la mediana e la moda dei 99 valori.

```
1  /* Fig. 6.16: fig06_16.c
2     Questo programma introduce l'argomento dell'analisi dei dati
      di un'indagine.
3     Esso calcola la media, la mediana e la moda dei dati */
4  #include <stdio.h>
5  #define SIZE 99
6
7  /* prototipi di funzione */
8  void mean( const int answer[] );
9  void median( int answer[] );
10 void mode( int freq[], const int answer[] );
11 void bubbleSort( int a[] );
```

Figura 6.16 Programma per l'analisi dei dati di un'indagine (continua)

```
12  void printArray( const int a[] );
13
14  /* l'esecuzione del programma inizia dalla funzione main */
15  int main()
16  {
17     int frequency[ 10 ] = { 0 }; /* inizializza il vettore frequency */
18
19     /* inizializza il vettore response */
20     int response[ SIZE ] =
21                        { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
22                          7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
23                          6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
24                          7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
25                          6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
26                          7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
27                          5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
28                          7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
29                          7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
30                          4, 5, 6, 1, 6, 5, 7, 8, 7 };
31
32     /* elabora le risposte */
33     mean( response );
34     median( response );
35     mode( frequency, response );
36
37     return 0; /* indica che il programma è terminato con successo */
38
39  } /* fine della funzione main */
40
41  /* calcola la media dei valori di tutte le risposte */
42  void mean( const int answer[] )
43  {
44     int j; /* contatore per calcolare il totale degli elementi
                  del vettore */
45    int total = 0; /* variabile per memorizzare la somma degli elementi
                        del vettore */
46
47     printf( "%s\n%s\n%s\n", "********", "  Mean", "******** );
48
49     /* calcola il totale dei valori delle risposte */
50     for ( j = 0; j < SIZE; j++ ) {
51        total += answer[ j ];
52     } /* fine del comando for */
53
54     printf( "The mean is the average value of the data\n"
55             "items. The mean is equal to the total of\n"
56             "all the data items divided by the number\n"
57             "of data items (%d). The mean value for\n"
58             "this run is: %d / %d = %.4f\n\n",
```

Figura 6.16 Programma per l'analisi dei dati di un'indagine (continua)

```
59            SIZE, total, SIZE, ( double ) total / SIZE );
60    } /* fine della funzione mean */
61
62    /* ordina il vettore e determina il valore dell'elemento mediano */
63    void median( int answer[] )
64    {
65       printf("\n%s\n%s\n%s\n%s",
66              "********", " Median", "********",
67              "The unsorted array of responses is");
68
69       printArray( answer ); /* visualizza il vettore non ordinato */
70
71       bubbleSort( answer ); /* ordina il vettore */
72
73       printf( "\n\nThe sorted array is" );
74       printArray( answer ); /* visualizza il vettore ordinato */
75
76       /* visualizza l'elemento mediano */
77       printf( "\n\nThe median is element %d of\n"
78              "the sorted %d element array.\n"
79              "For this run the median is %d\n\n",
80              SIZE / 2, SIZE, answer[ SIZE / 2 ] );
81    } /* fine della funzione median */
82
83    /* determina la risposta più frequente */
84    void mode( int freq[], const int answer[] )
85    {
86       int rating; /* contatore per accedere gli elementi 1-9 del vettore
                        freq */
87       int j; /* contatore per riassumere gli elementi 0-98 del vettore
                   answer */
88       int h; /* contatore per visualizzare gli istogrammi degli elementi
                  del vettore freq */
89       int largest = 0; /* rappresenta la frequenza massima */
90       int modeValue = 0; /* rappresenta la risposta più frequente */
91
92       printf( "\n%s\n%s\n%s\n",
93              "********", "  Mode", "******** " );
94
95       /* azzera le frequenze */
96       for ( rating = 1; rating <= 9; rating++ ) {
97          freq[ rating ] = 0;
98       } /* fine del comando for */
99
100      /* riassume le frequenze */
101      for ( j = 0; j < SIZE; j++) {
102         ++freq[ answer[ j ] ];
103      } /* fine del comando for */
104
```

Figura 6.16 Programma per l'analisi dei dati di un'indagine (continua)

```
105      /* visualizza le intestazioni per le colonne dei risultati */
106      printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
107              "Response", "Frequency", "Histogram",
108              "1    1    2    2", "5    0    5    0    5" );
109
110      /* visualizza i risultati */
111      for ( rating = 1; rating <= 9; rating++ ) {
112         printf( "%8d%11d          ", rating, freq[ rating ] );
113
114         /* tiene traccia del valore della moda e del valore massimo
               della frequenza */
115         if ( freq[ rating ] > largest ) {
116            largest = freq[ rating ];
117            modeValue = rating;
118         } /* fine del comando if */
119
120         /* visualizza la barra dell'istogramma che rappresenta
               il valore della frequenza */
121         for ( h = 1; h <= freq[ rating ]; h++ ) {
122            printf( "*" );
123         } /* fine del comando for interno */
124
125         printf( "\n" ); /* inizia una nuova linea di output */
126      } /* fine del comando for esterno */
127
128      /* visualizza il valore della moda */
129      printf( "The mode is the most frequent value.\n"
130              "For this run the mode is %d which occurred"
131              " %d times.\n", modeValue, largest );
132   } /* fine della funzione mode */
133
134   /* funzione che ordina un vettore per mezzo dell'algoritmo bubble
         sort */
135   void bubbleSort( int a[] )
136   {
137      int pass; /* contatore dei passaggi */
138      int j; /* contatore dei confronti */
139      int hold; /* locazione temporanea usata per scambiare gli elementi */
140
141      /* ciclo per controllare il numero dei passaggi */
142      for ( pass = 1; pass < SIZE; pass++ ) {
143
144         /* ciclo per controllare il numero di confronti per ogni
               passaggio */
145         for (j = 0; j < SIZE - 1; j++ ) {
146
147            /* scambia gli elementi se non sono in ordine */
148            if ( a[ j ] > a[ j+1 ] ) {
149               hold = a[ j ];
```

Figura 6.16 Programma per l'analisi dei dati di un'indagine (continua)

```
150               a[ j ] = a[ j+1 ];
151               a[ j+1 ] = hold;
152            } /* fine del comando if */
153
154         } /* fine del comando for interno */
155
156      } /* fine del comando for esterno */
157
158   } /* fine della funzione bubbleSort */
159
160   /* visualizza i contenuti del vettore (20 valori per riga) */
161   void printArray( int a[] )
162   {
163      int j; /* contatore */
164
165      /* visualizza i contenuti del vettore */
166      for ( j = 0; j < SIZE; j++ ) {
167
168         if ( j % 20 == 0 ) { /* inizia una nuova linea ogni 20 valori */
169            printf( "\n" );
170         } /* fine del comando if */
171
172         printf( "%2d", a[ j ] );
173      } /* fine del comando for */
174
175   } /* fine della funzione printArray */
```

Figura 6.16 Programma per l'analisi dei dati di un'indagine

La media è proprio quella aritmetica calcolata sui 99 valori. La funzione `mean` (riga 42) calcolerà la media sommando i 99 elementi e dividendo il risultato per 99.

La mediana è il "valore di mezzo". La funzione `median` (riga 63) determinerà la mediana richiamando `bubbleSort` (definita nella riga 135) per ordinare in modo ascendente il vettore delle opinioni e selezionare il valore di mezzo, `answer[ SIZE / 2 ]`, del vettore ordinato. Osservate che qualora ci fosse un numero pari di elementi, la mediana dovrebbe essere calcolata come la media dei due valori di mezzo. Al momento, la funzione `median` non dispone di questa facoltà. La funzione `printArray` (riga 161) sarà invocata per inviare in output il vettore `response`.

La moda è il valore che si presenta più frequentemente tra le 99 opinioni. La funzione `mode` (riga 84) determinerà la moda contando le occorrenze di ogni tipo di opinione e selezionando quella con il valore maggiore. Questa versione della funzione `mode` non gestirà un eventuale pareggio (consultate l'Esercizio 6.14). La funzione `mode` produrrà anche un istogramma per fornire un supporto grafico alla determinazione della moda. La Figura 6.17 contiene un esempio di esecuzione di questo programma. Questo esempio comprende la maggior parte delle comuni manipolazioni richieste solitamente dai problemi che utilizzano i vettori, incluso il loro passaggio alle funzioni.

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788


********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7


********
  Mode
********
Response  Frequency          Histogram

                                    1    1    2    2
                               5    0    5    0    5

       1          1          *
       2          3          ***
       3          4          ****
       4          5          *****
       5          8          ********
       6          9          *********
       7         23          ***********************
       8         27          ***************************
       9         19          *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Figura 6.17 Esempio di esecuzione del programma per l'analisi dei dati di un'indagine

6.8 La ricerca nei vettori

I programmatori lavoreranno spesso con grandi quantità di dati immagazzinati in vettori. A volte, potrà essere necessario determinare se un vettore contenga un elemento che corrisponda a un dato *valore chiave*. Il processo di ritrovamento di un particolare elemento in un vettore è detto *ricerca*. In questa sezione discuteremo due tecniche di ricerca: una semplice detta *ricerca lineare* e un'altra più efficiente (ma più complessa) detta *ricerca binaria*. Gli Esercizi 6.34 e 6.35, alla fine di questo capitolo, vi chiederanno di implementare le versioni ricorsive della ricerca lineare e di quella binaria.

Effettuare una ricerca in un vettore con una ricerca lineare

La ricerca lineare (Figura 6.18) confronta ogni elemento del vettore con la *chiave di ricerca*. Dato che il vettore non è in un ordine particolare, la probabilità che il valore ricercato sia ritrovato nel primo elemento sarà la stessa dell'ultimo. In media, quindi, il programma dovrà confrontare la chiave di ricerca con la metà degli elementi inclusi nel vettore.

```
1  /* Fig. 6.18: fig06_18.c
2     Ricerca lineare in un vettore */
3  #include <stdio.h>
4  #define SIZE 100
5
6  /* prototipo di funzione */
7  int linearSearch( const int array[], int key, int size );
8
9  /* l'esecuzione del programma inizia dalla funzione main */
10 int main()
11 {
12    int a[ SIZE ]; /* crea il vettore a */
13    int x; /* contatore per inizializzare gli elementi 0-99 del
                vettore a */
14    int searchKey; /* valore da trovare nel vettore a */
15    int element; /* variabile per memorizzare la posizione
                      di searchKey o -1 */
16
17    /* crea i dati */
18    for ( x = 0; x < SIZE; x++ ) {
19       a[ x ] = 2 * x;
20    } /* fine del comando for */
21
22    printf( "Enter integer search key:\n" );
23    scanf( "%d", &searchKey );
24
25    /* cerca di trovare searchKey nel vettore a */
26    element = linearSearch( a, searchKey, SIZE );
27
28    /* visualizza i risultati */
29    if ( element != -1 ) {
30       printf( "Found value in element %d\n", element );
31    } /* fine del ramo if */
```

Figura 6.18 Ricerca lineare in un vettore (continua)

```
32      else {
33         printf( "Value not found\n" );
34      } /* fine del ramo else */
35
36      return 0; /* indica che il programma è terminato con successo */
37
38   } /* fine della funzione main */
39
40   /* confronta la chiave con ogni elemento del vettore finché non viene
41      trovata la posizione o finché non viene raggiunta la fine del
42      vettore; restituisce l'indice dell'elemento se viene trovata la
        chiave, -1 altrimenti */
43   int linearSearch( const int array[], int key, int size )
44   {
45      int n; /* contatore */
46
47      /* esegue un ciclo scorrendo il vettore */
48      for ( n = 0; n < size; ++n) {
49
50         if ( array[ n ] == key ) {
51            return n; /* restituisce la posizione della chiave */
52         } /* fine del comando if */
53
54      } /* fine del comando for */
55
56      return -1; /* chiave non trovata */
57
58   } /* fine della funzione linearSearch */
```

```
Enter integer search key:
36
Found value in element 18


Enter integer search key:
37
Value not found
```

Figura 6.18 Ricerca lineare in un vettore

Effettuare una ricerca in un vettore con una ricerca binaria

Il metodo di ricerca lineare funziona bene con i vettori di piccole dimensioni o con quelli che non sono stati ordinati. Per i vettori di dimensioni ragguardevoli, invece, la ricerca lineare è inefficiente. Nel caso che il vettore sia ordinato, potrà essere utilizzata la tecnica notevolmente più veloce della ricerca binaria.

L'algoritmo della ricerca binaria, dopo ogni confronto, scarterà metà degli elementi in un vettore ordinato. L'algoritmo individua l'elemento di mezzo del vettore e lo confronta con la chiave di ricerca. Nel caso corrispondessero, la chiave di ricerca sarebbe stata individuata e

sarebbe restituito l'indice di quell'elemento. Nel caso non corrispondessero, il problema si ridurrebbe a una ricerca da eseguire in una delle due metà del vettore. La ricerca sarebbe eseguita nella prima metà del vettore, qualora la chiave di ricerca fosse inferiore al suo elemento di mezzo, in caso contrario, la ricerca sarebbe eseguita nella seconda metà del vettore. L'algoritmo sarebbe ripetuto su un quarto del vettore originale, qualora la chiave di ricerca non fosse stata ritrovata nel sottovettore specificato (il frammento del vettore originale). La ricerca continuerà finché la chiave non corrisponda all'elemento di mezzo di un sottovettore, o finché questo non sia formato da un unico elemento che non corrisponda alla chiave di ricerca (ovverosia, questa non ha alcuna corrispondenza nel vettore).

Nella peggiore delle ipotesi, eseguire una ricerca binaria in un vettore di 1024 elementi richiederà soltanto 10 confronti. Infatti, dividendo ripetutamente 1024 per 2 si otterrebbero i valori 512, 256, 128, 64, 32, 16, 8, 4, 2 e 1. Il numero 1024 (2^{10}) può essere diviso per 2 solo 10 volte per ottenere il valore 1. La divisione per due equivale a un confronto dell'algoritmo di ricerca binaria. Ne consegue quindi che per ritrovare la chiave di ricerca in un vettore di 1,048,576 (2^{20}) elementi sarebbero necessari 20 confronti al massimo. Per ritrovare la chiave di ricerca in un vettore di un miliardo di elementi sarebbero necessari 30 confronti al massimo. Questo è uno straordinario incremento di efficienza rispetto alla ricerca lineare, che richiede invece di eseguire un numero medio di confronti con la chiave di ricerca pari alla metà degli elementi presenti nel vettore. Per un vettore di un miliardo di elementi, si tratta della differenza tra una media di 500 milioni di confronti e un massimo di 30! Il numero massimo di confronti richiesti da un vettore potrà essere determinato, trovando la prima potenza di 2 che sia maggiore del numero di elementi presenti nel vettore.

La Figura 6.19 presenta la versione iterativa della funzione `binarySearch` (righe 45-77). Questa riceverà quattro argomenti: un vettore di interi `b` in cui effettuare la ricerca, un intero `searchKey` e gli indici `low` e `high` del vettore (questi ultimi definiscono la parte del vettore in cui effettuare la ricerca). Qualora la chiave di ricerca non corrisponda all'elemento di mezzo di un sottovettore, sarà modificato l'indice `low` o quello `high`, in modo tale che la ricerca possa essere eseguita su un sottovettore più piccolo. Nel caso che la chiave di ricerca sia inferiore all'elemento di mezzo, l'indice `high` sarà impostato con `middle - 1` e la ricerca sarà ripresa sugli elementi compresi tra `low` e `middle - 1`. Nel caso invece che la chiave di ricerca sia maggiore dell'elemento di mezzo, l'indice `low` sarà impostato con `middle + 1` e la ricerca sarà ripresa sugli elementi compresi tra `middle + 1` e `high`. Il programma utilizzerà un vettore di 15 elementi. La prima potenza di due superiore al numero degli elementi presenti in questo vettore sarà dunque 16 (24), perciò saranno necessari al massimo 4 confronti per ritrovare la chiave di ricerca. Questo programma utilizzerà la funzione `printHeader` (righe 80-99) per inviare in output gli indici del vettore, mentre userà la funzione `printRow` (righe 103-124) per inviare in output ogni sottovettore durante il processo di ricerca binaria. L'elemento di mezzo di ogni sottovettore sarà marcato con un asterisco (*), in modo da indicare quello con il quale sarà confrontata la chiave di ricerca.

```
1  /* Fig. 6.19: fig06_19.c
2     Ricerca binaria in un vettore */
3  #include <stdio.h>
4  #define SIZE 15
5
```

Figura 6.19 Ricerca binaria in un vettore ordinato (continua)

```
 6   /* prototipi di funzione */
 7   int binarySearch( const int b[], int searchKey, int low, int high );
 8   void printHeader( void );
 9   void printRow( const int b[], int low, int mid , int high );
10
11   /* l'esecuzione del programma inizia dalla funzione main */
12   int main()
13   {
14      int a[ SIZE ]; /* crea il vettore a */
15      int i; /* contatore per inizializzare gli elementi 0-14
                  del vettore a */
16      int key; /* valore da trovare nel vettore a */
17      int result; /* valore per memorizzare la posizione di key
                       oppure -1 */
18
19      /* crea i dati */
20      for ( i = 0; i < SIZE; i++ ) {
21         a[ i ] = 2 * i;
22      } /* fine del comando for */
23
24      printf( "Enter a number between 0 and 28: " );
25      scanf( "%d", &key );
26
27      printHeader();
28
29      /* cerca key nel vettore a */
30      result = binarySearch( a, key, 0, SIZE - 1 );
31
32      /* visualizza i risultati */
33      if ( result != -1 ) {
34         printf( "\n%d found in array element %d\n", key, result );
35      } /* fine del ramo if */
36      else {
37         printf( "\n%d not found\n", key );
38      } /* fine del ramo else */
39
40      return 0; /* indica che il programma è terminato con successo */
41
42   } /* fine della funzione main */
43
44   /* funzione per effettuare una ricerca binaria in un vettore */
45   int binarySearch( const int b[], int searchKey, int low, int high )
46   {
47      int middle; /* variabile per memorizzare l'elemento di mezzo
                       del vettore */
48
49     /* esegue un ciclo finché l'indice low non è più grande dell'indice
          high */
```

Figura 6.19 Ricerca binaria in un vettore ordinato (continua)

```
50      while ( low <= high ) {
51
52         /* determina l'elemento di mezzo del sottovettore in cui si
              effettua la ricerca */
53         middle = ( low + high ) / 2;
54
55         /* visualizza il sottovettore utilizzato in questa iterazione
              del ciclo */
56         printRow( b, low, middle, high );
57
58         /* se searchKey coincide con l'elemento di mezzo, restituisce
              middle */
59         if ( searchKey == b[ middle ] ) {
60            return middle;
61         } /* fine del ramo if */
62
63       /* se searchKey è minore dell'elemento di mezzo, imposta il nuovo
            valore di high */
64         else if (searchKey < b[middle]) {
65            high = middle - 1; /* ricerca nella parte "bassa"
                                    del vettore */
66         } /* fine del ramo else if */
67
68         /* se searchKey è maggiore dell'elemento di mezzo, imposta
              il nuovo valore di low */
69         else {
70            low = middle + 1; /* ricerca nella parte "alta"
                                   del vettore */
71         } /* fine del ramo else */
72
73      } /* fine del comando while */
74
75      return -1;   /* searchKey non trovata */
76
77   } /* fine della funzione binarySearch */
78
79   /* Visualizza un'intestazione per l'output */
80   void printHeader( void )
81   {
82      int i; /* contatore */
83
84      printf( "\nSubscripts:\n" );
85
86      /* visualizza l'intestazione delle colonne */
87      for ( i = 0; i < SIZE; i++) {
88         printf( "%3d ", i );
89      } /* fine del comando for */
90
91      printf( "\n" ); /* comincia una nuova linea di output */
```

Figura 6.19 Ricerca binaria in un vettore ordinato (continua)

```
92
93       /* visualizza linee composte da caratteri - */
94       for ( i = 1; i <= 4 * SIZE; i++ ) {
95          printf( "-" );
96       } /* fine del comando for */
97
98       printf( "\n" ); /* comincia una nuova linea di output */
99    } /* fine della funzione printHeader */
100
101   /* Visualizza una riga di output che mostra la parte
102      del vettore che è in corso di elaborazione. */
103   void printRow( const int b[], int low, int mid, int high )
104   {
105      int i; /* contatore per iterare scorrendo il vettore b */
106
107      /* esegue un ciclo scorrendo interamente il vettore */
108      for ( i = 0; i < SIZE; i++ ) {
109
110         /* visualizza degli spazi se ci si trova al di fuori
                dell'intervallo del sottovettore corrente */
111         if ( i < low || i > high ) {
112            printf( "   " );
113         } /* fine del ramo if */
114         else if ( i == mid ) { /* visualizza l'elemento di mezzo */
115            printf( "%3d*", b[ i ] );  /* marca il valore di mezzo */
116         } /* fine del ramo else if */
117         else { /* visualizza gli altri elementi nel sottovettore */
118            printf( "%3d ", b[ i ] );
119         } /* fine del ramo else */
120
121      } /* fine del comando for */
122
123      printf( "\n" ); /* comincia una nuova linea di output */
124   } /* fine della funzione printRow */
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                              16  18  20  22* 24  26  28
                                              24  26* 28
                                              24*

25 not found
```

Figura 6.19 Ricerca binaria in un vettore ordinato (continua)

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                     8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 ---------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

Figura 6.19 Ricerca binaria in un vettore ordinato

6.9 I vettori multidimensionali

I vettori in C possono anche avere più di un indice. Un utilizzo tipico dei vettori multidimensionali è la rappresentazione di *tabelle* di valori formate da informazioni organizzate in *righe* e *colonne*. Per identificare un particolare elemento della tabella, dovremo specificare due indici: il primo identificherà per convenzione la riga dell'elemento, mentre il secondo (sempre per convenzione) ne identificherà la colonna. Le tabelle o i vettori che per identificare un particolare elemento richiedono due indici sono detti *vettori bidimensionali o matrici*. Osservate che i vettori multidimensionali possono avere più di due indici.

La Figura 6.20 illustra una matrice `a`. Questa conterrà tre righe e quattro colonne, perciò si usa definirla matrice 3 per 4. In generale, una matrice con *m* righe e *n* colonne sarà detta *matrice m per n*. Ogni elemento nella matrice `a` è identificato nella Figura 6.20 da un nome di elemento avente la forma `a[ i ][ j ]`; `a` rappresenta il nome della matrice, mentre `i` e `j` sono gli indici che identificano univocamente ogni elemento incluso in `a`. Osservate che i nomi degli elementi nella prima riga hanno tutti `0` come primo indice; mentre i nomi degli elementi nella quarta colonna hanno tutti `3` come secondo indice.

Errore tipico 6.9

Fare riferimento all'elemento di una matrice usando la forma `a[x, y]` *invece di* `a[x][y]`.

Un vettore multidimensionale potrà essere inizializzato contestualmente alla propria dichiarazione, in un modo molto simile a quello utilizzato per un vettore unidimensionale. Per esempio, una matrice `b[2][2]` potrà essere dichiarata e inizializzata con

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

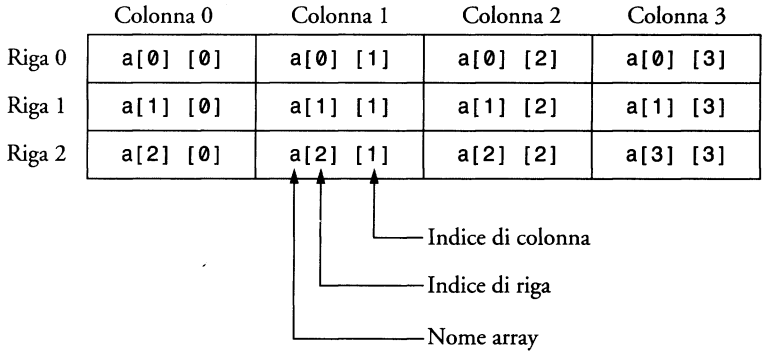


Figura 6.20 Una matrice con tre righe e quattro colonne

I valori saranno raggruppati per riga all'interno di parentesi graffe. I valori nel primo insieme di parentesi inizializzeranno la riga 0 e i valori nel secondo insieme di parentesi inizializzeranno la riga 1. Perciò i valori 1 e 2 serviranno per inizializzare gli elementi `b[ 0 ][ 0 ]` e `b[ 0 ][ 1 ]`, rispettivamente, mentre i valori 3 e 4 inizializzeranno gli elementi `b[ 1 ][ 0 ]` e `b[ 1 ][ 1 ]`, rispettivamente. Nell'eventualità che per una data riga non sia stato fornito un numero sufficiente di inizializzatori, gli elementi rimanenti di quella riga saranno inizializzati a `0`. Quindi, la dichiarazione

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

inizializzerà `b[ 0 ][ 0 ]` a 1, `b[ 0 ][ 1 ]` a 0, `b[ 1 ][ 0 ]` a 3 e `b[ 1 ][ 1 ]` a 4.

La Figura 6.21 mostra la definizione e l'inizializzazione di alcune matrici.

```
1  /* Fig. 6.21: fig06_21.c
2     Inizializzare i vettori multidimensionali */
3  #include <stdio.h>
4
5  void printArray( const int a[][ 3 ] ); /* prototipo di funzione */
6
7  /* l'esecuzione del programma inizia dalla funzione main */
8  int main()
9  {
10    /* inizializza array1, array2, array3 */
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    printf( "Values in array1 by row are:\n" );
16    printArray( array1 );
17
18    printf( "Values in array2 by row are:\n" );
19    printArray( array2 );
20
21    printf( "Values in array3 by row are:\n" );
```

Figura 6.21 Inizializzare i vettori multidimensionali (continua)

```
22      printArray( array3 );
23
24      return 0; /* indica che il programma è terminato con successo */
25
26   } /* fine della funzione main */
27
28   /* funzione per visualizzare un vettore con due righe e tre colonne */
29   void printArray( const int a[][ 3 ] )
30   {
31      int i; /* contatore delle righe */
32      int j; /* contatore delle colonne */
33
34      /* esegue un ciclo scorrendo le righe */
35      for ( i = 0; i <= 1; i++ ) {
36
37         /* visualizza i valori delle colonne */
38         for ( j = 0; j <= 2; j++ ) {
39            printf( "%d ", a[ i ][ j ] );
40         } /* fine del comando for interno */
41
42         printf( "\n" ); /* inizia una nuova linea di output */
43      } /* fine del comando for esterno */
44
45   } /* fine della funzione printArray */
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Figura 6.21 Inizializzare i vettori multidimensionali

Il programma dichiarerà tre matrici di due righe e tre colonne (ovverosia sei elementi a testa). La dichiarazione di `array1` (riga 11) fornisce sei inizializzatori in due sottoliste. La prima sottolista inizializzerà la prima riga (ovvero la riga 0) della matrice con i valori 1, 2 e 3; mentre la seconda sottolista inizializzerà la seconda riga (ovvero la riga 1) della matrice con i valori 4, 5 e 6. Qualora dall'elenco degli inizializzatori di `array1` fossero rimosse le parentesi graffe che racchiudono ognuna delle sottoliste, il compilatore provvederebbe a inizializzare prima gli elementi della prima riga e, solo in seguito, quelli della seconda riga. La dichiarazione di `array2` (riga 12) fornirà cinque inizializzatori. Questi saranno assegnati alla prima riga e quindi alla seconda. Ogni elemento che non avrà un inizializzatore esplicito sarà inizializzato automaticamente a zero, perciò `array2[ 1 ][ 2 ]` sarà azzerato.

La dichiarazione di `array3` (riga 13) fornirà tre inizializzatori suddivisi in due sottoliste. Quella destinata alla prima riga inizializzerà esplicitamente i suoi due primi elementi con 1 e

2. Il terzo elemento sarà invece azzerato automaticamente. La sottolista destinata alla seconda riga inizializzerà esplicitamente il suo primo elemento a 4. Gli ultimi due elementi saranno azzerati automaticamente.

Il programma invocherà quindi la funzione `printArray` (righe 29-45) per inviare in output ogni elemento della matrice. Osservate che la definizione della funzione specifica il parametro della matrice come `const int a[][ 3 ]`. Le parentesi quadre di un vettore unidimensionale sono vuote, nella lista dei parametri di una funzione che la riceve come argomento. Non è obbligatorio neanche il primo indice di un vettore multidimensionale, mentre lo sono tutti quelli successivi. Il compilatore utilizzerà questi indici per determinare le posizioni in memoria degli elementi dei vettori multidimensionali. Tutti gli elementi dei vettori saranno immagazzinati in memoria in modo consecutivo indipendentemente dal numero degli indici. La prima riga di una matrice immagazzinata in memoria sarà seguita dalla seconda riga.

Fornendo i valori degli indici, contestualmente alla dichiarazione del parametro, consentiremo al compilatore di poter indicare alla funzione il modo per individuare un qualsiasi elemento del vettore. Nelle matrici, ogni riga è fondamentalmente un vettore unidimensionale. Durante l'accesso alla matrice, il compilatore dovrà conoscere esattamente quanti elementi ci sono in ogni riga, così che possa saltare la quantità appropriata di posizioni di memoria per individuare un elemento in una riga particolare. Di conseguenza, il compilatore saprà che, quando nel nostro esempio dovrà accedere ad `a[ 1 ][ 2 ]`, per giungere alla seconda riga (la riga 1), dovrà saltare in memoria i tre elementi della prima riga. Solo allora il compilatore potrà accedere al terzo elemento di quella riga (l'elemento 2).

Molte manipolazioni tipiche delle matrici utilizzano il comando di ripetizione `for`. Per esempio, la struttura seguente azzererà tutti gli elementi della terza riga della matrice `a` nella Figura 6.20:

```
for ( column = 0; column <= 3; column++ )
   a[ 2 ][ column ] = 0;
```

Abbiamo specificato la *terza riga*, di conseguenza sappiamo che il primo indice sarà sempre `2` (di nuovo, `0` è la prima riga, mentre `1` è la seconda). Il ciclo `for` farà dunque variare solo il secondo indice (ovverosia quello della colonna). Il precedente comando `for` è equivalente alle istruzioni di assegnamento:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

Il seguente comando `for` nidificato determinerà il totale di tutti gli elementi presenti nella matrice `a`.

```
total = 0;

for ( row = 0; row <= 2; row++ )
   for ( column = 0; column <= 3; column++ )
      total += a[ row ][ column ];
```

Il comando `for` sommerà gli elementi della matrice una riga per volta. Il comando `for` più esterno incomincerà impostando `row` (ovverosia l'indice della riga) a `0`, così che la prima riga possa essere sommata dal comando `for` più interno. Il comando `for` più esterno incrementerà `row` a `1`, così che possano essere sommati gli elementi della seconda riga. Il comando `for` più

esterno incrementerà `row` a 2, così che possano essere sommati gli elementi della terza riga. Il risultato sarà visualizzato quando il comando `for` nidificato avrà terminato la propria esecuzione.

Il programma della Figura 6.22 eseguirà molte altre manipolazioni tipiche sulla matrice 3 per 4 `studentGrades` utilizzando dei comandi `for`. Ogni riga della matrice rappresenta uno studente, mentre ogni colonna rappresenta una votazione in uno dei quattro esami che lo studente ha affrontato durante il semestre. Le manipolazioni della matrice saranno eseguite da quattro funzioni. La funzione `minimum` (righe 44-66) determinerà il voto più basso di ogni studente nel semestre. La funzione `maximum` (righe 69-91) determinerà il voto più alto di ogni studente nel semestre. La funzione `average` (righe 94-106) determinerà la media di un particolare studente nel semestre. La funzione `printArray` (righe 109-130) invierà in output la matrice bidimensionale in un formato tabulare ordinato.

Ognuna delle funzioni `minimum`, `maximum` e `printArray` riceverà tre argomenti: la matrice `studentGrades` (chiamata `grades` in ogni funzione), il numero degli studenti (le righe della matrice) e il numero degli esami (le colonne della matrice). Ogni funzione itererà sulla matrice `grades` utilizzando dei comandi `for` nidificati. Il seguente comando `for` nidificato è tratto dalla definizione della funzione `minimum`:

```
/* esegue un ciclo scorrendo le righe di grades */
   for ( i = 0; i < pupils; i++ ) {

      /* esegue un ciclo scorrendo le colonne di grades */
      for ( j = 0; j < tests; j++ ) {

         if ( grades[ i ][ j ] < lowGrade ) {
            lowGrade = grades[ i ][ j ];

         } /* fine del comando if */

      } /* fine del comando for interno */

   } /* fine del comando for esterno */
```

Il comando `for` più esterno incomincerà impostando `i` (ovverosia l'indice delle righe) a `0`, così che gli elementi della prima riga (ovverosia, i voti del primo studente) possano essere confrontati con la variabile `lowGrade` nel corpo del comando `for` più interno. Il comando `for` più interno itererà a sua volta sulle quattro votazioni di una riga particolare e le confronterà con `lowGrade`. Nel caso che un voto sia inferiore a `lowGrade`, questa ne assumerà il valore. Il comando `for` più esterno incrementerà l'indice di riga a 1. Gli elementi della seconda riga saranno confrontati con la variabile `lowGrade`. Il comando `for` più esterno incrementerà quindi l'indice di riga a 2. Gli elementi della terza riga saranno confrontati con la variabile `lowGrade`. Quando l'esecuzione della struttura nidificata sarà stata completata, `lowGrade` conterrà il voto più piccolo presente nella matrice. La funzione `maximum` lavorerà in modo simile a `minimum`.

```
1  /* Fig. 6.22: fig06_22.c
2     Esempio di matrice */
3  #include <stdio.h>
4  #define STUDENTS 3
5  #define EXAMS 4
6
```

Figura 6.22 Esempio di utilizzo delle matrici (continua)

```
/* prototipi di funzione */
int minimum( const int grades[][ EXAMS ], int pupils, int tests );
int maximum( const int grades[][ EXAMS ], int pupils, int tests );
double average( const int setOfGrades[], int tests );
void printArray( const int grades[][ EXAMS ], int pupils, int tests );

/* l'esecuzione del programma inizia dalla funzione main */
int main()
{
   int student; /* contatore degli studenti */

   /* inizializza i voti degli studenti per tre studenti (righe) */
   const int studentGrades[ STUDENTS ][ EXAMS ] =
                                    { { 77, 68, 86, 73 },
                                      { 96, 87, 89, 78 },
                                      { 70, 90, 86, 81 } };

   /* visualizza il vettore studentGrades */
   printf( "The array is:\n" );
   printArray( studentGrades, STUDENTS, EXAMS );

   /* determina il minimo e il massimo valore dei voti */
   printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
          minimum( studentGrades, STUDENTS, EXAMS ),
          maximum( studentGrades, STUDENTS, EXAMS ) );

   /* calcola il voto medio per ogni studente */
   for ( student = 0; student < STUDENTS; student++ ) {
      printf( "The average grade for student %d is %.2f\n",
            student, average( studentGrades[ student ], EXAMS ) );
   } /* fine del comando for */

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */

/* Trova il voto minimo */
int minimum( const int grades[][ EXAMS ], int pupils, int tests )
{
   int i; /* contatore degli studenti */
   int j; /* contatore degli esami */
   int lowGrade = 100; /* inizializza con il voto più alto possibile */

   /* esegue un ciclo scorrendo le righe di grades */
   for ( i = 0; i < pupils; i++ ) {

      /* esegue un ciclo scorrendo le colonne di grades */
      for ( j = 0; j < tests; j++ ) {

         if ( grades[ i ][ j ] < lowGrade ) {
            lowGrade = grades[ i ][ j ];
```

Figura 6.22 Esempio di utilizzo delle matrici (continua)

```
58            } /* fine del comando if */
59
60         } /* fine del comando for interno */
61
62      } /* fine del comando for esterno */
63
64      return lowGrade; /* restituisce il voto minimo */
65
66   } /* fine della funzione minimum */
67
68   /* Trova il voto massimo */
69   int maximum( const int grades[][ EXAMS ], int pupils, int tests )
70   {
71      int i; /* contatore degli studenti */
72      int j; /* contatore degli esami */
73     int highGrade = 0; /* inizializza con il voto più basso possibile */
74
75      /* esegue un ciclo scorrendo le righe di grades */
76      for ( i = 0; i < pupils; i++ ) {
77
78         /* esegue un ciclo scorrendo le colonne di grades */
79         for ( j = 0; j < tests; j++ ) {
80
81            if ( grades[ i ][ j ] > highGrade ) {
82               highGrade = grades[ i ][ j ];
83            } /* fine del comando if */
84
85         } /* fine del comando for interno */
86
87      } /* fine del comando for esterno */
88
89      return highGrade; /* restituisce il voto massimo */
90
91   } /* fine della funzione maximum */
92
93   /* Determina il voto medio per un particolare studente */
94   double average( const int setOfGrades[], int tests )
95   {
96      int i; /* contatore degli esami */
97      int total = 0; /* somma dei voti degli esami */
98
99      /* calcola il totale di tutti i voti di uno studente */
100      for ( i = 0; i < tests; i++ ) {
101         total += setOfGrades[ i ];
102      } /* fine del comando for */
103
104      return ( double ) total / tests; /* media */
105
106   } /* fine della funzione average */
107
108   /* Visualizza la matrice */
```

Figura 6.22 Esempio di utilizzo delle matrici (continua)

```
109  void printArray( const int grades[][ EXAMS ], int pupils, int tests )
110  {
111     int i; /* contatore degli studenti */
112     int j; /* contatore degli esami */
113
114     /* visualizza le intestazioni delle colonne */
115     printf( "                [ 0 ]  [ 1 ]  [ 2 ]  [ 3 ]" );
116
117     /* visualizza i voti in formato tabulare */
118     for ( i = 0; i < pupils; i++ ) {
119
120        /* visualizza l'etichetta per la riga */
121        printf( "\nstudentGrades[ %d ] ", i );
122
123        /* visualizza i voti per uno studente */
124        for ( j = 0; j < tests; j++ ) {
125           printf( "%-5d", grades[ i ][ j ] );
126        } /* fine del comando for interno */
127
128     } /* fine del comando for esterno */
129
130  } /* fine della funzione printArray */
```

```
The array is:
                [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Figura 6.22 Esempio di utilizzo delle matrici

La funzione `average` (riga 63) riceverà due argomenti: il vettore unidimensionale `setOfGrades`, che conterrà i risultati degli esami per un dato studente, e il numero dei risultati di esame presenti all'interno del vettore. Quando `average` sarà invocata, il primo argomento passato sarà `studentGrades[ student ]`. Con questa istruzione sarà passato ad `average` l'indirizzo di una riga della matrice. L'argomento `studentGrades[ 1 ]` corrisponderà all'indirizzo di partenza della seconda riga della matrice. Ricordate, infatti, che una matrice è fondamentalmente un vettore di vettori unidimensionali e che il nome di un vettore unidimensionale corrisponde al suo indirizzo di memoria. La funzione `average` calcolerà la somma degli elementi inclusi nel vettore, dividerà il totale ottenuto per il numero degli esiti degli esami e restituirà il risultato in virgola mobile.

Esercizi di autovalutazione

6.1 Rispondete a ognuna delle seguenti domande:

a) Le liste e le tabelle di valori sono immagazzinate nei _____.

b) Gli elementi di un vettore sono correlati dal fatto che hanno lo stesso _____ e _____.

c) Il numero utilizzato per far riferimento a un particolare elemento di un vettore è il suo _____.

d) Per specificare la dimensione di un vettore, dovrebbe essere utilizzata una _____, perché renderebbe il programma più scalabile.

e) Il processo di messa in ordine degli elementi contenuti in un vettore è detto _____ del vettore.

f) Determinare se un vettore contenga un certo valore chiave è detto _____ nel vettore.

g) Un vettore che utilizzi due indici è detto vettore _____.

6.2 Stabilite se le seguenti affermazioni siano vere o false. Qualora la risposta sia falsa, spiegatene il motivo.

a) Un vettore può immagazzinare valori di molti tipi differenti.

b) L'indice di un vettore può essere del tipo di dato `double`.

c) Nel caso che in una lista di inizializzatori ci siano meno elementi di quelli presenti nel vettore, il C inizializzerebbe automaticamente quelli rimanenti utilizzando l'ultimo valore incontrato nella lista degli inizializzatori.

d) È un errore inserire in una lista di inizializzatori più elementi di quanti siano presenti nel vettore.

e) Qualora un singolo elemento di un vettore sia passato a una funzione e sia modificato dalla stessa, nella funzione chiamante quell'elemento conterrà il valore modificato.

6.3 Rispondete alle seguenti domande riguardanti un vettore chiamato `fractions`.

a) Definite una costante simbolica `SIZE` che possa essere rimpiazzata dal testo di sostituzione 10.

b) Dichiarate un vettore con `SIZE` elementi di tipo `double` e inizializzateli a `0`.

c) Puntate al quarto elemento dall'inizio del vettore.

d) Fate riferimento all'elemento 4 del vettore.

e) Assegnate il valore `1,667` all'elemento nove del vettore.

f) Assegnate il valore `3,333` al settimo elemento del vettore.

g) Visualizzate gli elementi 6 e 9 del vettore, con due cifre di precisione a destra della virgola decimale e mostrate l'output che sarà effettivamente visualizzato sullo schermo.

h) Visualizzate tutti gli elementi del vettore usando il comando di ripetizione `for`. Supponete che sia stata definita la variabile intera `x` per controllare il ciclo. Mostrate l'output.

6.4 Scrivete dei comandi per portare a termine quanto segue:

a) Dichiarate la matrice `table` di tipo intero con 3 righe e 3 colonne. Supponete che sia stata definita la costante simbolica `SIZE` con testo di sostituzione 3.

b) Quanti elementi conterrà la matrice `table`? Visualizzate il numero totale di elementi.

c) Utilizzate un comando di ripetizione `for`, per inizializzare ogni elemento della matrice con la somma dei propri indici. Supponete che come variabili di controllo siano state dichiarate `x` e `y`.

d) Visualizzate i valori di ogni elemento della matrice `table`. Supponete che la matrice sia stata inizializzata con la dichiarazione:

```
int table[ SIZE ][ SIZE ] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

6.5 Trovate l'errore in ognuno dei seguenti segmenti di programma e correggetelo.

a)
```
#define SIZE 100;
```

b)
```
SIZE = 10;
```

c) Assumete
```
int b[ 10 ] = { 0 }, i;

for ( i = 0; i <= 10; i++ )
   b[ i ] = 1;
```

d)
```
#include <stdio.h>;
```

e) Assumete
```
int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };

a[ 1, 1 ] = 5;
```

f)
```
#define VALUE = 120
```

Risposte agli esercizi di autovalutazione

6.1 a) Vettori. b) Nome, tipo. c) Indice. d) Costante simbolica. e) Ordinamento. f) Ricerca. g) Bidimensionale.

6.2 a) Falso. Un vettore può immagazzinare soltanto valori dello stesso tipo.
b) Falso. L'indice di un vettore deve essere un intero o una espressione intera.
c) Falso. Il C azzererà automaticamente gli elementi rimanenti.
d) Vero.
e) Falso. I singoli elementi di un vettore sono passati per valore. Le modifiche apportate all'interno della funzione chiamata sarebbero riscontrabili nel vettore originale, solo se alla funzione fosse stato passato l'intero vettore.

6.3 a)
```
#define SIZE 10
```
b)
```
double fractions[ SIZE ] = { 0 };
```
c)
```
fractions[ 3 ]
```
d)
```
fractions[ 4 ]
```
e)
```
fractions[ 9 ] = 1.667;
```
f)
```
fractions[ 6 ] = 3.333;
```
g)
```
printf("%.2f %.2f\n", fractions[ 6 ], fractions[ 9 ]);
```
Output: `3.33 1.67.`

h)
```
for ( x = 0; x < SIZE; x++ )

   printf( "fractions[%d] = %f\n", x, fractions[ x ] );
```

Output:

```
fractions[0] = 0.000000
fractions[1] = 0.000000
fractions[2] = 0.000000
fractions[3] = 0.000000
fractions[4] = 0.000000
fractions[5] = 0.000000
fractions[6] = 3.333000
fractions[7] = 0.000000
fractions[8] = 0.,000000
fractions[9] = 1.,667000
```

6.4 a)
```
int table[ SIZE ][ SIZE ];
```

b) Nove elementi. `printf( "%d\n", SIZE * SIZE );`

c)
```
for (x = 0; x < SIZE; x++)
```

```
   for (y = 0; y < SIZE; y++)
      table[ x ][ y ] = x + y;
```

d)
```
for (x = 0; x < SIZE; x++)

   for (y = 0; y < SIZE; y++)
      printf( "table[%d][%d] = %d\n", x, y, table[ x ][ y ] );
```

Output:

```
table[0][0] = 1
table[0][1] = 8
table[0][2] = 0
table[1][0] = 2
table[1][1] = 4
table[1][2] = 6
table[2][0] = 5
table[2][1] = 0
table[2][2] = 0
```

6.5 a) Errore: il punto e virgola alla fine della direttiva `#define` del preprocessore.
Correzione: eliminate il punto e virgola.

b) Errore: assegnamento di un valore a una costante simbolica utilizzando un'istruzione di assegnamento.
Correzione: assegnate il valore della costante simbolica in una direttiva `#define` del preprocessore, senza utilizzare l'operatore di assegnamento, come in `#define SIZE 10`.

c) Errore: riferimento a un elemento esterno ai limiti del vettore (`b[ 10 ]`).
Correzione: modificate a `9` il valore finale della variabile di controllo.

d) Errore: un punto e virgola alla fine della direttiva `#include` del preprocessore.
Correzione: eliminate il punto e virgola.

e) Errore: indice della matrice scritto in modo scorretto.
Correzione: cambiare l'istruzione in `a[ 1 ][ 1 ] = 5;`

f) Errore: assegnare un valore a una costante simbolica usando un comando di assegnamento.
Correzione: assegnare un valore alla costante simbolica in una direttiva `#define` del preprocessore senza utilizzare l'operatore di assegnamento, come ad esempio in `#define VALUE 120`.

Esercizi

6.6 Riempite gli spazi in ognuna delle seguenti righe:

a) Il C immagazzina le liste di valori nei _____.

b) Gli elementi di un vettore sono correlati dal fatto che _____.

c) Quando si fa riferimento a un elemento di un vettore, il numero di posizione contenuto all'interno delle parentesi quadre è detto _____.

d) I nomi dei cinque elementi del vettore `p` sono _____, _____, _____, _____ e _____.

e) Il contenuto di un particolare elemento di un vettore è il _____ dell'elemento.

f) Dare un nome a un vettore, stabilirne il tipo e specificare il numero degli elementi contenuti nello stesso è la _____ del vettore.

g) Il processo di sistemazione in ordine ascendente o discendente degli elementi contenuti in un vettore è chiamato _____.

h) In una matrice, il primo indice (per convenzione) identifica la _____ di un elemento, mentre il secondo indice (per convenzione) identifica la _____ di un elemento.

i) Una matrice m per n contiene _____ righe, _____ colonne e _____ elementi.
j) Il nome dell'elemento nella riga 3 e nella colonna 5 di una matrice `d` è _____.

6.7 Stabilite quali delle seguenti affermazioni siano vere e quali false; per quelle false, spiegate perché lo sono.

a) Per far riferimento a una particolare posizione o elemento all'interno di un vettore, specificheremo il nome di questo e il valore di quel particolare elemento.
b) La dichiarazione di un vettore gli riserverà uno spazio in memoria.
c) Per indicare che debbano essere riservate 100 posizioni per il vettore di interi `p`, il programmatore scriverà la dichiarazione

```
p[ 100 ];
```

d) Un programma C che azzeri i 15 elementi di un vettore dovrà contenere un'istruzione `for`.
e) Un programma C che sommi gli elementi di una matrice dovrà contenere delle istruzioni `for` nidificate.
f) La media, la mediana e la moda del seguente gruppo di valori sono rispettivamente 5, 6 e 7: 1, 2, 5, 6, 7, 7, 7.

6.8 Scrivete delle istruzioni che eseguano ognuna delle seguenti attività:

a) Visualizzate il valore contenuto nel settimo elemento del vettore di caratteri `f`.
b) Prendete in input un valore e sistematelo nell'elemento 4 del vettore unidimensionale di valori in virgola mobile `b`.
c) Inizializzate a `8` ognuno dei 5 elementi interi contenuti nel vettore unidimensionale `g`.
d) Sommate i 100 elementi contenuti nel vettore di valori in virgola mobile `c`.
e) Copiate il vettore `a` nella prima porzione del vettore `b`. Assumete che sia già stata scritta la dichiarazione `double a[ 11 ], b[ 34 ];`
f) Determinate e visualizzate il valore più piccolo e quello più grande contenuti nel vettore `w` di 99 elementi in virgola mobile.

6.9 Considerate la matrice di interi `t` di dimensioni 2 per 5.

a) Scrivete una dichiarazione per `t`.
b) Quante righe avrà `t`?
c) Quante colonne avrà `t`?
d) Quanti elementi avrà `t`?
e) Scrivete i nomi di tutti gli elementi contenuti nella seconda riga di `t`.
f) Scrivete i nomi di tutti gli elementi contenuti nella terza colonna di `t`.
g) Scrivete una singola istruzione che azzeri gli elementi di `t` contenuti nella riga 1 e nella colonna 2.
h) Scrivete una serie di istruzioni che azzerino ogni elemento di `t`. Non utilizzate una struttura di ripetizione.
i) Scrivete una struttura `for` nidificata che azzeri ogni elemento di `t`.
j) Scrivete un'istruzione che prenda in input dal terminale i valori per gli elementi di `t`.
k) Scrivete una serie di istruzioni che determinino e visualizzino il valore più piccolo contenuto nella matrice `t`.
l) Scrivete un'istruzione che visualizzi gli elementi contenuti nella prima riga di `t`.
m) Scrivete un'istruzione che sommi gli elementi contenuti nella quarta colonna di `t`.
n) Scrivete una serie di istruzioni che visualizzino la matrice `t` in un formato tabulare ordinato. Utilizzate gli indici delle colonne come intestazioni da porre in cima alla tabella e, allo stesso modo, visualizzate gli indici delle righe a sinistra di ognuna di esse.

6.10 Utilizzate un vettore unidimensionale per risolvere il seguente problema. Un'azienda retribuisce i suoi venditori con delle provvigioni. Un venditore riceve $ 200 la settimana più il 9 per cento delle proprie vendite lorde portate a termine in quella settimana. Per esempio, un venditore che faccia incassare $ 3000 di venduto lordo, in una settimana, riceverà $ 200 più il 9 per cento di $ 3000, ovverosia un totale di $ 470. Scrivete un programma C, utilizzando un vettore di contatori che determini quanti venditori abbiano guadagnato una retribuzione compresa in ognuno dei seguenti intervalli (supponete che la retribuzione di ogni venditore sia troncata a una somma intera):

a) $ 200-$ 299
b) $ 300-$ 399
c) $ 400-$ 499
d) $ 500-$ 599
e) $ 600-$ 699
f) $ 700-$ 799
g) $ 800-$ 899
h) $ 900-$ 999
i) $ 1000 e oltre

6.11 L'ordinamento a bolle presentato nella Figura 6.15 è inefficiente per vettori di grandi dimensioni. Apportate le seguenti semplici modifiche in modo da migliorare l'efficienza dell'ordinamento a bolle.

a) Dopo il primo passaggio, il numero più grande sarà stato sicuramente sistemato nell'elemento con l'indice più alto del vettore; dopo il secondo passaggio, i due numeri più grandi saranno "a posto" e così via. Invece di eseguire nove confronti a ogni passaggio, modificate l'ordinamento a bolle così che esegua otto confronti nel secondo passaggio, sette nel terzo e così via.
b) I dati nel vettore potrebbero già essere nell'ordine appropriato o in uno vicino a questo, perciò perché eseguire nove passaggi quando ne potrebbero bastare meno? Modificate l'ordinamento a bolle, in modo da verificare se alla fine di ogni passaggio siano stati eseguiti degli scambi. Nel caso che non ne siano stati eseguiti, allora i dati saranno già nell'ordine appropriato, perciò il programma dovrà terminare la propria esecuzione. Nel caso invece che siano stati eseguiti degli scambi, sarà necessario almeno un altro passaggio.

6.12 Scrivete delle istruzioni singole che eseguano ognuna delle seguenti operazioni su vettori unidimensionali:

a) Azzerate i 10 elementi del vettore di interi `counts`.
b) Aggiungete 1 a ognuno dei 15 elementi contenuti nel vettore di interi `bonus`.
c) Leggete dalla tastiera i 12 valori in virgola mobile del vettore `monthlyTemperatures`.
d) Visualizzate in colonna i 5 valori contenuti nel vettore di interi `bestScores`.

6.13 Trovate l'errore in ognuna delle seguenti istruzioni:

a) Assumete: `char str[ 5 ];`

```
scanf( "%s", str );   /* L'utente digita hello */
```

b) Assumete: `int a[ 3 ];`

```
printf( "$d  %d  %d\n", a[ 1 ], a[ 2 ], a[ 3 ] );
```

c) `double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`
d) Assumete: `d[ 2 ][ 10 ];`

```
d[ 1, 9 ] = 2.345;
```

6.14 Modificate il programma della Figura 6.16, in modo che la funzione `mode` sia in grado di gestire un pareggio per il valore della moda. Modificate anche la funzione `median`, in modo che sia calcolata la media dei due elementi di mezzo, in un vettore che contenga un numero pari di elementi.

6.15 Utilizzate un vettore unidimensionale per risolvere il seguente problema. Leggete in input 20 numeri, ognuno dei quali sarà compreso tra 10 e 100, estremi inclusi. Ogni volta che leggete un numero, visualizzatelo qualora non sia un duplicato di uno già letto. Preparatevi al "caso peggiore" in cui tutti i 20 numeri siano differenti. Utilizzate il vettore più piccolo possibile per risolvere questo problema.

6.16 Etichettate gli elementi della matrice 3 per 5 `sales`, in modo da indicare l'ordine in cui essi saranno azzerati dal seguente segmento di programma:

```
for ( row = 0; row <= 2; row++ )
   for ( column = 0; column <= 4; column++ )
      sales[ row ][ column ] = 0;
```

6.17 Che cosa farà il seguente programma?

```
/* ex06_17.c */
/* Che cosa farà questo programma? */
#include <stdio.h>
#define SIZE 10

int whatIsThis( const int b[], int p ); /* prototipo di funzione */

/* l'esecuzione del programma inizia dalla funzione main */
int main()
{
   int x; /* memorizza il valore di ritorno della funzione whatIsThis */

   /* inizializza il vettore a */
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   x = whatIsThis( a, SIZE );

   printf( "Result is %d\n", x );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */

/* Che cosa farà questa funzione? */
int whatIsThis( const int b[], int size )
{
   /* caso base */
   if ( size == 1 ) {
      return b[ 0 ];
   } /* fine del ramo if */
   else { /* passo ricorsivo */

      return b[ size - 1 ] + whatIsThis( b, size - 1 );
   } /* fine del ramo else */

} /* fine della funzione whatIsThis */
```

6.18 Che cosa farà il seguente programma?

```
/* ex06_18.c */
/* Che cosa farà questo programma? */
#include <stdio.h>
#define SIZE 10

```

```
/* prototipo di funzione */
void someFunction( const int b[], int startIndex, int size );

/* l'esecuzione del programma inizia dalla funzione main */
int main()
{
   int a[ SIZE ] = { 8, 3, 1, 2, 6, 0, 9, 7, 4, 5 }; /* inizializza a */

   printf( "Answer is:\n" );
   someFunction( a, 0, SIZE );
   printf( "\n" );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */

/* Che cosa farà questa funzione? */
void someFunction( const int b[], int startIndex, int size )
{
   if ( startIndex < size ) {
      someFunction( b, startIndex + 1, size);
      printf( "%d  ", b[ startIndex ] );
   } /* fine del ramo if */

} /* fine della funzione someFunction */
```

6.19 Scrivete un programma che simuli il lancio di due dadi. Il programma dovrà utilizzare `rand` per lanciare il primo dado e invocarla nuovamente per lanciare il secondo dado. Dovrà quindi essere calcolata la somma dei due valori. [*Nota*: poiché ogni dado può mostrare un valore intero compreso tra 1 a 6, la somma dei due valori potrà variare tra 2 e 12, con 7 come somma più frequente e 2 e 12 come somme meno frequenti.] La Figura 6.23 mostra le 36 possibili combinazioni dei due dadi. Il vostro programma dovrà lanciare i due dadi 36.000 volte. Utilizzate un vettore unidimensionale per sommare il numero di occorrenze di ogni somma possibile. Visualizzate i risultati in un formato tabulare. Determinate anche se i totali sono sensati: ci sono sei modi per ottenere un 7, perciò circa un sesto dei lanci dovrebbe ottenerlo.

6.20 Scrivete un programma che esegua 1000 volte (senza intervento umano) il gioco dei dadi (craps) e rispondete a ognuna delle seguenti domande:

a) Quanti giochi sono stati vinti al primo lancio, al secondo, ... al ventesimo lancio e dopo il ventesimo?

b) Quanti giochi sono stati persi al primo lancio, al secondo, ... al ventesimo lancio e dopo il ventesimo?

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 6.23 I risultati del lancio dei dadi

c) Quali sono le possibilità di vittoria al gioco dei dadi? [*Nota*: sappiate che il gioco dei dadi è uno dei più equi tra quelli da casinò. A quale conclusione vi porta questa considerazione?]
d) Qual è la lunghezza media di un gioco a dadi?
e) Le probabilità di vittoria aumentano con la lunghezza del gioco?

6.21 (*Sistema di prenotazione per linee aeree*) Una piccola compagnia aerea ha appena comprato un computer per il suo nuovo sistema di prenotazione automatica. Il presidente vi ha chiesto di programmare il nuovo sistema. Voi dovete scrivere un programma che assegni i posti su ogni volo dell'unico aereo dell'aerolinea (capacità: 10 posti).

Il vostro programma dovrà visualizzare il seguente menu di scelte:

```
Please type 1 for "first class"
Please type 2 for "economy"
```

Nel caso che il cliente digiti 1, allora il vostro programma dovrà assegnare un posto nella sezione prima classe (quelli da 1 a 5). Nel caso che il cliente digiti 2, allora il vostro programma dovrà assegnare un posto nella sezione economica (quelli da 6 a 10). Il vostro programma dovrà quindi stampare una carta di imbarco, che dovrà indicare il numero di posto assegnato al passeggero e se quello si trova nella sezione prima classe o economica dell'aeroplano.

Utilizzate un vettore unidimensionale per rappresentare la mappa dei posti sull'aereo. Azzerate tutti gli elementi del vettore in modo da indicare che tutti i posti sono vuoti. Man mano che i posti saranno stati assegnati, imposterete a 1 l'elemento corrispondente del vettore in modo da indicare che il posto non è più disponibile.

Naturalmente, il vostro programma non dovrà mai assegnare un posto che sia già stato assegnato. Quando la sezione prima classe sarà piena, il vostro programma dovrà chiedere al cliente se sia disposto ad accettare una sistemazione nella sezione economica (e viceversa). Eseguite l'appropriata assegnazione di posto, qualora la sua risposta sia affermativa. Visualizzate il messaggio `"Next flight leaves in 3 hours"` (`"Il prossimo volo decollerà fra 3 ore"`), qualora la sua risposta sia negativa.

6.22 Utilizzate una matrice per risolvere il seguente problema. Un'azienda ha quattro venditori (numerati da 1 a 4) che vendono cinque differenti prodotti (numerati da 1 a 5). Una volta al giorno, ognuno dei venditori fornisce un tagliando per ogni tipo di prodotto venduto. Ogni tagliando contiene:

1. Il numero del venditore
2. Il numero del prodotto
3. Il valore totale, espresso in dollari, del venduto giornaliero di quel prodotto

Di conseguenza, ogni venditore fornisce tra 0 e 5 tagliandi al giorno. Supponete che siano disponibili i dati dei tagliandi dell'ultimo mese. Scrivete un programma che legga le suddette informazioni, riguardanti il venduto dell'ultimo mese, e sommi le vendite totali per venditore e per prodotto. Tutti i totali dovranno esse immagazzinati nella matrice `sales`. Dopo avere elaborato tutte le informazioni dell'ultimo mese, visualizzate i risultati in formato tabulare in modo che le colonne rappresentino i vari venditori e le righe rappresentino i singoli prodotti. Sommate ogni riga, in modo da ottenere le vendite totali dell'ultimo mese per ognuno dei prodotti; sommate ogni colonna, in modo da ottenere il totale delle vendite dell'ultimo mese per ognuno dei venditori. La vostra stampa tabulare dovrà includere i suddetti totali a destra delle righe e in fondo alle colonne.

6.23 (*Turtle Graphics*) Il linguaggio Logo, particolarmente popolare tra gli utenti di personal computer, ha reso famoso il concetto di turtle graphics (grafici della tartaruga). Immaginate una tartaruga meccanica che girovaga in una stanza sotto il controllo di un programma C. La tartaruga ha una penna in una delle due posizioni, alzata o abbassata. La tartaruga traccia delle linee man mano che si muove con la penna abbassata; quando questa è alzata, si muove liberamente senza scrivere niente. In questo problema, simulerete le operazioni della tartaruga e creerete, allo stesso tempo, un blocchetto per gli schizzi.

Utilizzate una matrice `floor` 50 per 50 e azzeratela. Leggete i comandi da un vettore che li contenga. In ogni momento, mantenete traccia della posizione corrente della tartaruga e dello stato (alzata o abbassata) della penna. Supponete che la tartaruga cominci sempre dalla posizione 0,0 del pavimento con la penna alzata. L'insieme dei comandi per la tartaruga che il vostro programma dovrà elaborare è elencato in Figura 6.24.

Comando	Significato
1	Alza la penna
2	Abbassa la penna
3	Gira a destra
4	Gira a sinistra
5,10	Vai in avanti di 10 spazi (o un numero diverso da 10)
6	Visualizza la matrice 50 per 50
9	Fine dei dati (valore sentinella)

Figura 6.24 I comandi della tartaruga

Supponete che la tartaruga sia in una qualche posizione vicina al centro del pavimento. Il seguente "programma" disegnerebbe e visualizzerebbe un quadrato 12 per 12:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

Man mano che la tartaruga si muoverà con la penna abbassata, imposterete gli elementi corrispondenti della matrice `floor` a `1`. Nel momento in cui il programma avrà impartito il comando `6` (visualizza), stamperete un asterisco, o qualsiasi altro carattere abbiate scelto, in corrispondenza di ogni `1` incluso nella matrice. Stamperete invece uno spazio in corrispondenza di ogni zero. Scrivete un programma C che implementi le capacità grafiche della tartaruga discusse in questo esercizio. Scrivete diversi programmi di turtle graphics, in modo che siano disegnate delle forme interessanti. Aggiungete altri comandi per incrementare la potenza del vostro linguaggio di turtle graphics.

6.24 (*Il Giro del Cavallo*) Uno dei problemi più interessanti per gli appassionati del gioco degli scacchi è quello del Giro del Cavallo, proposto originariamente dal matematico Eulero. La questione è questa: può il pezzo degli scacchi chiamato cavallo muoversi in una scacchiera vuota, visitando una sola volta ognuna delle 64 caselle? Studieremo approfonditamente questo problema affascinante.

Il cavallo compie delle mosse a forma di L: due caselle in una direzione e una in una direttrice perpendicolare a quella precedente. Ne consegue che, da una casella centrale di una scacchiera vuota, il cavallo potrà eseguire otto differenti mosse (numerate da 0 a 7), come mostrato dalla Figura 6.25.

a) Disegnate una scacchiera 8 per 8 su un foglio di carta e tentate di portare a termine manualmente un giro del cavallo. Inserirete un 1 nella prima casella in cui vi muoverete, un 2

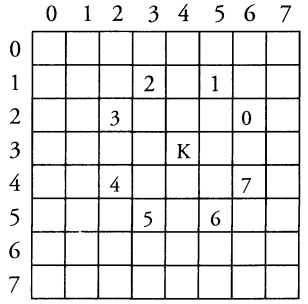


Figura 6.25 Le otto possibili mosse del cavallo

nella seconda, un 3 nella terza ecc. Prima di incominciare il giro, provate a fare una stima delle mosse che riuscirete a compiere, ricordando che un giro completo consiste di 64 mosse. Dove siete arrivati? Vi siete avvicinati alla vostra stima?

b) Ora svilupperemo un programma che muoverà il cavallo in giro per la scacchiera. La scacchiera stessa sarà rappresentata dalla matrice 8 per 8 `board`. Ognuna delle caselle sarà inizialmente azzerata. Descriveremo ognuna delle otto possibili mosse scindendole nelle loro componenti orizzontali e verticali. Per esempio, una mossa come la 0 mostrata nella Figura 6.25 potrà essere scomposta in un movimento orizzontale (due caselle a destra) e in uno spostamento verticale (una casella in alto). La mossa 2 consisterà di uno spostamento orizzontale (una casella a sinistra) e di uno spostamento verticale (due caselle in alto). Le mosse orizzontali a sinistra e quelle verticali verso l'alto saranno indicate con dei numeri negativi. Le otto mosse potranno essere descritte da due vettori unidimensionali, `horizontal` e `vertical`, come segue:

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

Le variabili `currentRow` e `currentColumn` indicheranno rispettivamente la riga e la colonna della posizione corrente del cavallo. Per eseguire una mossa di tipo `moveNumber`, dove `moveNumber` sia compreso tra 0 e 7, il vostro programma utilizzerà queste istruzioni:

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Utilizzate un contatore che vari nell'intervallo da `1` a `64`. Registrate l'ultimo valore assunto dal contatore di ogni casella visitata dal cavallo. Ricordate di verificare ogni mossa poten-

ziale per assicurarvi che il cavallo non abbia già visitato quella casella. E, naturalmente, verificate ogni mossa potenziale per assicurarvi che il cavallo non cada fuori della scacchiera. Ora scrivete un programma che porti il cavallo in giro per la scacchiera. Eseguite il programma. Quante mosse ha compiuto il cavallo?

c) Dopo aver tentato di scrivere ed eseguire un programma che risolva il problema del Giro del Cavallo, avrete probabilmente sviluppato qualche preziosa intuizione. Utilizzeremo proprio quelle per sviluppare un metodo euristico (ovverosia una strategia) per muovere il cavallo. I metodi euristici non garantiscono il successo ma, se sviluppati accuratamente, aumentano enormemente le probabilità di successo. Dovreste aver notato che le caselle più esterne sono, in un certo senso, più problematiche di quelle più vicine al centro della scacchiera. Infatti, le caselle più problematiche, o inaccessibili, sono proprio quelle dei quattro angoli.

L'intuizione potrà suggerirvi che, al principio, sarà meglio tentare di muovere il cavallo nelle caselle più problematiche, lasciando libere quelle più facili da raggiungere, in modo da avere maggiori probabilità di successo verso la fine del giro, quando la scacchiera sarà congestionata.

Potremo sviluppare una "euristica dell'accessibilità", classificando ognuna delle caselle in base alla loro accessibilità e movendo conseguentemente il cavallo nella casella meno accessibile tra quelle raggiungibili dal suo particolare movimento a L. Etichetteremo la matrice `accessibility` con dei valori che indichino, per ogni casella, il numero di quelle dalle quali essa potrà essere raggiunta. In una scacchiera vuota, le caselle del centro saranno dunque valutate `8`, quelle degli angoli `2`, mentre le altre avranno dei valori d'accessibilità pari a `3`, `4` o `6`, come segue:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Ora scrivete una versione del programma per il Giro del Cavallo che utilizzi l'euristica dell'accessibilità. In ogni momento, il cavallo dovrà muoversi nella casella con il numero di accessibilità più basso. In caso di parità tra caselle, il cavallo potrà muoversi verso una qualsiasi di quelle a pari merito. Di conseguenza, il giro potrà cominciare in uno qualsiasi dei quattro angoli. [*Nota*: il vostro programma dovrà ridurre i valori di accessibilità poiché con lo spostamento del cavallo sulla scacchiera aumenterà il numero di caselle già visitate. In questo modo, in un qualsiasi momento del giro, il numero di accessibilità di ogni casella rimarrà esattamente uguale al numero di caselle dalle quali essa potrà essere raggiunta.] Eseguite questa versione del vostro programma. Avete ottenuto un giro completo? Modificate ora il programma facendo in modo che il cavallo compia 64 giri, partendo da ognuna delle caselle della scacchiera. Quanti giri completi avete ottenuto?

d) Scrivete una versione del programma del Giro del Cavallo che, qualora si imbatta in un pareggio tra due o più caselle, decida quale scegliere dopo aver dato uno sguardo alle caselle che potranno essere raggiunte da quelle a "pari merito". Il vostro programma dovrà muovere il cavallo nella casella da cui sarà possibile raggiungere quella con il valore di accessibilità più basso.

6.25 (*Il Giro del Cavallo: approcci brutali*) Nell'Esercizio 6.24 abbiamo sviluppato una soluzione per il problema del Giro del Cavallo. L'approccio utilizzato, detto "euristica dell'accessibilità", genera molte soluzioni ed è efficiente.

Man mano che la potenza dei computer continuerà ad aumentare, saremo in grado di risolvere molti problemi sfruttando solamente le loro capacità di calcolo e degli algoritmi meno sofisticati. In genere questo tipo di approccio alla soluzione dei problemi è detto "brutale o con la forza bruta".

a) Utilizzate un generatore di numeri casuali per consentire al cavallo di girovagare a caso per la scacchiera, naturalmente rispettando il suo legittimo incedere a L. Il vostro programma dovrà eseguire un giro e visualizzare la scacchiera risultante. Quanta strada ha percorso il cavallo?

b) Il programma precedente avrà prodotto molto probabilmente un giro relativamente breve. Modificate ora il vostro programma in modo da tentare 1000 giri. Utilizzate un vettore unidimensionale per conservare le occorrenze delle varie lunghezze ottenute da ogni giro. Nel momento in cui avrà completato i suoi 1000 tentativi, il vostro programma dovrà visualizzare le suddette informazioni in un formato tabulare ordinato. Qual è stato il risultato migliore?

c) Il programma precedente vi avrà fornito molto probabilmente qualche giro "degno di nota", ma nessun giro completo. Ora "lasciatevi andare" e lasciate andare anche il vostro programma, finché non riesca a produrre un giro completo. [*Attenzione*: questa versione del programma potrebbe durare per ore anche su un computer potente.] Ancora una volta, conservate in una tabella le occorrenze delle varie lunghezze ottenute da ogni giro e visualizzatela, quando sarà stato trovato il primo giro completo. Quanti tentativi ha compiuto il vostro programma, prima di produrre un giro completo? Quanto tempo ha impiegato?

d) Confrontate la versione brutale del Giro del Cavallo con quella che ha utilizzato l'euristica dell'accessibilità. Quale versione ha richiesto uno studio più accurato del problema? Quale algoritmo è stato più difficile da sviluppare? Quale ha richiesto più potenza di elaborazione? Sarà possibile avere a priori la certezza di ottenere un giro completo, usando l'approccio dell'euristica dell'accessibilità? Sarà possibile avere a priori la certezza di ottenere un giro completo, usando l'approccio con la forza bruta? Argomentate i pro e i contro della risoluzione dei problemi utilizzando la forza bruta in generale.

6.26 (*Le otto Regine*) Un altro problema per gli appassionati del gioco degli scacchi è quello delle otto Regine. Detto semplicemente: è possibile sistemare otto regine su una scacchiera vuota in modo che nessuna di loro "attacchi" le altre, ovverosia in modo che non ci siano due regine sulla stessa riga, sulla medesima colonna o lungo la stessa diagonale? Utilizzate il tipo di ragionamento sviluppato nell'Esercizio 6.24 per formulare un metodo euristico che risolva il problema delle otto Regine. Eseguite il vostro programma. [*Suggerimento*: sarà possibile assegnare un valore numerico a ogni casella della scacchiera, in modo da indicare il numero di quelle che saranno "escluse", in una scacchiera vuota, una volta che una regina sarà stata sistemata in quella casella. Per esempio, a ognuno dei quattro angoli sarebbe assegnato il valore 22, come mostrato nella Figura 6.26.)]

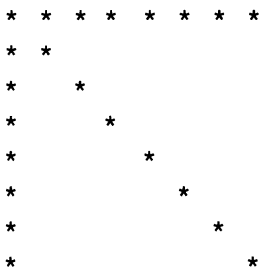


Figura 6.26 Le 22 caselle escluse sistemando una regina nell'angolo in alto a sinistra

Una volta che questi "numeri di esclusione" saranno stati sistemati in tutte le 64 caselle, un metodo euristico appropriato potrà essere: sistemate la prossima regina nella casella con il numero di eliminazione più basso. Per quale motivo questa strategia è intuitivamente attraente?

6.27 (*Le otto Regine: approcci brutali*) In questo esercizio svilupperete diversi approcci brutali per risolvere il problema delle otto Regine introdotto nell'Esercizio 6.26.

a) Risolvete il problema delle otto Regine, utilizzando la tecnica della forza bruta casuale sviluppata nell'Esercizio 6.25.
b) Usate una tecnica esaustiva: tentate cioè tutte le possibili combinazioni per il posizionamento delle otto regine sulla scacchiera.
c) Per quale motivo pensate che l'approccio esaustivo con la forza bruta non sia appropriato per risolvere il problema del Giro del Cavallo?
d) Confrontate e contrastate i due approcci brutali, casuale e esaustivo, in generale.

6.28 (*Eliminazione dei duplicati*) Nel Capitolo 12 esploreremo le strutture di dati con ricerca binaria ad alta velocità. Una caratteristica degli alberi di ricerca binaria è che i valori duplicati saranno scartati, nello stesso momento in cui gli elementi saranno inseriti nell'albero. Questa caratteristica è detta appunto eliminazione dei duplicati. Scrivete un programma che produca 20 numeri casuali compresi tra 1 e 20. Il programma dovrà immagazzinare in un vettore tutti i valori non duplicati. Utilizzate il vettore più piccolo possibile per portare a termine questo compito.

6.29 (*Il Giro del Cavallo: verifica del giro ciclico*) Nel Giro del Cavallo, definiremo completo un giro in cui il cavallo avrà effettuato 64 mosse, toccando ogni casella della scacchiera una sola volta. Definiremo ciclico un giro completo in cui la 64° posizione disti una mossa da quella in cui il cavallo ha cominciato il giro. Modificate il programma del Giro del Cavallo che avete scritto nell'Esercizio 6.24, in modo da verificare se un giro completo sia anche ciclico.

6.30 (*Il Crivello di Eratostene*) Un numero primo è qualsiasi intero che possa essere diviso soltanto per se stesso e per 1. Il Crivello di Eratostene è appunto un metodo per trovare i numeri primi. Esso funziona come segue:

1) Create un vettore con tutti gli elementi inizializzati a 1 (vero). Gli elementi del vettore i cui indici corrispondano a un numero primo resteranno a 1. Al termine della elaborazione, tutti gli altri elementi del vettore saranno azzerati.
2) Partendo dall'indice 2 del vettore (l'indice 1 è primo per definizione), qualora il valore dell'elemento puntato sia 1, scorrete il resto del vettore e azzerate gli elementi i cui indici siano multipli di quello dell'elemento puntato. Nel caso dell'indice 2, saranno azzerati tutti gli elementi successivi del vettore con indici che siano multipli di 2 (ovverosia gli indici 4, 6, 8, 10 ecc.). Nel caso dell'indice 3, saranno azzerati tutti gli elementi successivi del vettore con indici che siano multipli di 3 (ovverosia gli indici 6, 9, 12, 15 ecc.).

Nel momento in cui questo processo sarà stato completato, gli elementi del vettore che conterranno ancora il valore uno indicheranno che il loro indice è un numero primo e, di conseguenza, che potrà essere visualizzato. Scrivete un programma che utilizzi un vettore di 1000 elementi per determinare e visualizzare i numeri primi compresi tra 1 e 999. Ignorate l'elemento 0 del vettore.

6.31 (*Bucket Sort*) Un bucket sort incomincia dichiarando un vettore unidimensionale di interi positivi, quello che dovrà essere ordinato, e una matrice di interi con le righe indicizzate da 0 a 9 e le colonne da 0 a $n - 1$, dove n è il numero dei valori presenti nel vettore che dovrà essere ordinato. Ogni riga della matrice è detta bucket (secchio). Scrivete una funzione `bucketSort` che riceva come argomenti un vettore di interi e la sua dimensione.

L'algoritmo è il seguente:

1) Scorrete il vettore unidimensionale e sistemate ognuno dei suoi valori in una riga della matrice, basandovi sulla cifra delle unità. Per esempio, il 97 sarà sistemato nella riga 7, il 3 sarà inserito nella riga 3 e il 100 sarà posto nella riga 0.

2) Scorrete la matrice e riportate i valori nel vettore originario. Nel vettore unidimensionale il nuovo ordine dei suddetti valori sarà: 100, 3 e 97.
3) Ripetete questo processo per ogni successiva posizione delle cifre (decine, centinaia, migliaia ecc.) e fermatevi quando sarà stata elaborata la cifra più significativa del numero più grande.

Al termine del secondo passaggio sul vettore, il 100 sarà sistemato nella riga 0, il 3 sarà inserito nella riga 0 (poiché ha solo una cifra) e il 97 sarà posto nella riga 9. L'ordine dei valori nel vettore unidimensionale sarà a questo punto: 100, 3 e 97. Al termine del terzo passaggio, il 100 sarà sistemato nella riga 1, il 3 e il 97 saranno inseriti nella riga 0 (il 97 dopo il 3). Il bucket sort garantisce che tutti i valori saranno ordinati appropriatamente, dopo che sarà stata elaborata la cifra più significativa del numero più grande. Il bucket sort saprà di avere terminato il proprio compito, quando tutti i valori saranno stati ricopiati nella riga zero della matrice.

Osservate che la matrice avrà una dimensione pari a dieci volte quella del vettore di interi da ordinare. Questa tecnica è più efficiente di quella dell'ordinamento a bolle, ma richiede una capacità di memoria notevolmente superiore. L'ordinamento a bolle richiede solo una posizione di memoria in più per il tipo di dato da ordinare. Il bucket sort è un esempio di compromesso tra consumo di memoria ed efficienza: è vero che utilizza più memoria, ma è anche vero che è più efficiente. Questa versione del bucket sort richiede che per ogni passaggio i dati siano riportati nel vettore originale. Si potrebbe anche creare una seconda matrice e spostare ripetutamente i dati tra le due matrici, finché non siano stati ricopiati tutti nella riga zero di una delle due. A quel punto la riga zero conterrebbe il vettore ordinato.

Esercizi sulla ricorsione

6.32 (Ordinamento per selezione) Un ordinamento per selezione ricercherà in un vettore l'elemento più piccolo e, quando l'avrà individuato, lo scambierà di posto con il primo elemento del vettore. Il processo si ripeterà per il sottovettore che comincia con il secondo elemento del vettore. Ogni passaggio sul vettore farà in modo che un elemento sia sistemato nella posizione appropriata. Questo ordinamento richiede capacità di elaborazione simili a quelle dell'ordinamento a bolle: per un vettore di n elementi, dovranno essere eseguiti n - 1 passaggi e, per ogni sottovettore, dovranno essere compiuti $n - 1$ confronti per individuare il valore più piccolo. Il vettore risulterà ordinato nel momento in cui il sottovettore da elaborare conterrà un solo elemento. Scrivete una funzione ricorsiva `selectionSort` che implementi questo algoritmo.

6.33 (*Palindromi*) Un palindromo è una stringa che si legge allo stesso modo da sinistra a destra e da destra a sinistra. Alcuni esempi di palindromi sono: "radar", "able was i ere i saw elba" e "a man a plan a canal panama". Scrivete una funzione ricorsiva `testPalindrome` che restituisca 1, qualora la stringa immagazzinata nel vettore sia un palindromo, e 0 in caso contrario. La funzione dovrà ignorare gli spazi e la punteggiatura eventualmente presenti nella stringa.

6.34 (*Ricerca lineare*) Modificate il programma della Figura 6.18, in modo da utilizzare una funzione ricorsiva `linearSearch` che esegua una ricerca lineare all'interno del vettore. La funzione dovrà ricevere come argomenti un vettore di interi e la sua dimensione. Nel caso che la chiave di ricerca sia stata ritrovata, restituirete l'indice del vettore, altrimenti restituirete –1.

6.35 (*Ricerca binaria*) Modificate il programma della Figura 6.19, in modo che utilizzi una funzione ricorsiva `binarySearch` che esegua una ricerca binaria all'interno del vettore. La funzione dovrà ricevere, come argomenti, un vettore di interi e gli indici di inizio e di fine del sottovettore in cui dovrà essere eseguita la ricerca. Nel caso che la chiave di ricerca sia stata ritrovata, restituirete l'indice del vettore, altrimenti restituirete –1.

6.36 (*Le otto Regine*) Modificate il programma delle otto Regine che avete creato nell'Esercizio 6.26 così che risolva il problema in modo ricorsivo.

6.37 (*Visualizzare un vettore*) Scrivete una funzione ricorsiva `printArray` che riceva come argomenti un vettore e la sua dimensione e non restituisca alcun dato. La funzione dovrà terminare la propria elaborazione e restituire il controllo a quella chiamante, quando avrà ricevuto un vettore di dimensione 0.

6.38 (*Visualizzare una stringa al contrario*) Scrivete una funzione ricorsiva `stringReverse` che riceva come argomento un vettore di caratteri, lo visualizzi al contrario (dall'ultimo carattere al primo) e non restituisca nessun dato. La funzione dovrà terminare la propria elaborazione e restituire il controllo a quella chiamante, quando avrà incontrato il carattere nullo di terminazione della stringa.

6.39 (*Trovare il valore minimo in un vettore*) Scrivete una funzione ricorsiva `recursiveMinimum` che riceva come argomenti un vettore di interi e la sua dimensione e restituisca l'elemento più piccolo del vettore. La funzione dovrà terminare la propria elaborazione e restituire il controllo a quella chiamante, quando avrà ricevuto un vettore che contenga un solo elemento.

Capitolo 7

I puntatori in C

Obiettivi

- Capire i puntatori e gli operatori sui puntatori.
- Essere in grado di utilizzare i puntatori per simulare il passaggio per riferimento degli argomenti delle funzioni.
- Comprendere la stretta correlazione tra i puntatori, i vettori e le stringhe.
- Comprendere l'utilizzo dei puntatori a funzioni.
- Essere in grado di dichiarare e utilizzare i vettori di stringhe.

7.1 Introduzione

In questo capitolo, discuteremo una delle più potenti caratteristiche del linguaggio di programmazione C: i *puntatori*. Questa è una delle caratteristiche del C più difficili da padroneggiare. I puntatori consentono ai programmi di simulare le chiamate per riferimento e di creare e manipolare le strutture di dati dinamiche, ovverosia quelle che possono crescere e decrescere durante l'esecuzione di un programma, come le *liste concatenate*, le code, le pile e gli alberi. Questo capitolo spiegherà i concetti fondamentali dei puntatori. Il Capitolo 10 invece esaminerà l'utilizzo dei puntatori con le strutture. Il Capitolo 12 introdurrà le tecniche di *gestione dinamica della memoria* e presenterà degli esempi che creeranno e utilizzeranno le strutture di dati dinamiche.

7.2 Dichiarazione e inizializzazione dei puntatori

I puntatori sono delle variabili che come valore assumono degli indirizzi di memoria. Di norma una variabile contiene direttamente un valore specifico. Un puntatore invece contiene l'indirizzo di una variabile nella quale è immagazzinato quel valore specifico. In questo senso, il nome di una variabile fa *direttamente* riferimento a un valore, mentre un puntatore lo fa *indirettamente* (Figura 7.1). Per questo motivo il riferimento a un valore per mezzo di un puntatore è detto *deriferimento*.

I puntatori, come ogni altra variabile, devono essere dichiarati prima di poter essere utilizzati. La dichiarazione

```
int *countPtr, count;
```

dichiarerà la variabile `countPtr` di tipo `int *` (ovverosia, un puntatore a un valore intero) e si leggerà "`countPtr` è un puntatore a `int`" o "`countPtr` punta a un oggetto di tipo `int`". Anche la variabile `count` sarà dichiarata di tipo `int`, ma non sarà un puntatore a `int`. L'asterisco (`*`) presente nella dichiarazione sarà applicato solo a `countPtr`. Un `*` che sia stato

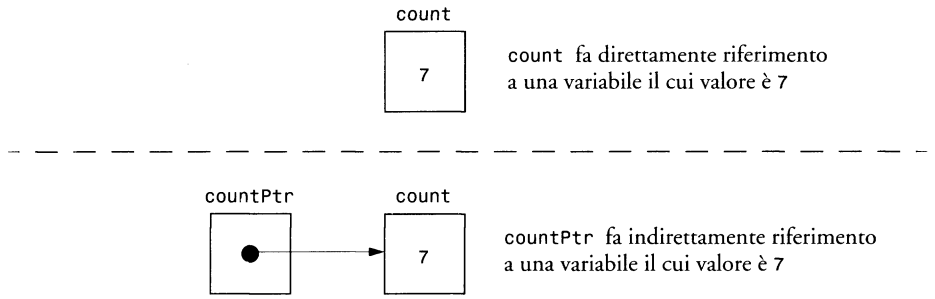


Figura 7.1 Far riferimento a una variabile in modo diretto e indiretto

utilizzato in una dichiarazione nel modo che abbiamo visto, indicherà che la variabile dichiarata sarà un puntatore. I puntatori potranno essere dichiarati in modo da fare riferimento a oggetti di qualsiasi tipo di dato.

Errore tipico 7.1

*La notazione che fa uso dell'asterisco * per dichiarare variabili di tipo puntatore non sarà applicata a tutti i nomi di variabile che siano stati inclusi in una dichiarazione. Ogni puntatore dovrà essere dichiarato con un * che preceda il suo nome; ad esempio, se volete dichiarare* `xPtr` *e* `yPtr` *come puntatori a* `int`*, dovrete usare* `int *xPtr, *yPtr;`.

Buona abitudine 7.1

Includete le lettere `ptr` *nei nomi per le variabili di tipo puntatore, così da rendere evidente che siano dei puntatori e che sarà necessario manipolarli in modo appropriato.*

I puntatori dovrebbero essere inizializzati e ciò potrà essere fatto o contestualmente alla loro dichiarazione o con un'istruzione di assegnamento. Un puntatore potrà essere inizializzato con `0`, `NULL` o con un indirizzo. Un puntatore che contenga il valore `NULL` non farà riferimento a nessun dato. `NULL` è una costante simbolica definita nel file di intestazione `<stddef.h>` (che viene incluso da parecchi altri file di intestazione, come ad esempio `<stdio.h>`). L'inizializzazione di un puntatore con `0` è equivalente a quella eseguita con il valore `NULL`, ma è preferibile utilizzare quest'ultimo. Ciò perché uno `0` che sia stato assegnato a un puntatore sarà prima convertito in un puntatore di tipo appropriato. Lo `0` è l'unico valore intero che potrà essere assegnato direttamente a una variabile di tipo puntatore. L'assegnamento dell'indirizzo di una variabile a un puntatore sarà discusso nella Sezione 7.3.

Collaudo e messa a punto 7.1

Inizializzate i puntatori in modo da prevenire dei risultati inattesi.

7.3 Gli operatori sui puntatori

L'ampersand (`&`, e commerciale), od *operatore di indirizzo*, è un operatore unario che restituisce l'indirizzo del suo operando. Per esempio, assumendo le dichiarazioni

```
int y = 5;
int *yPtr;
```

l'istruzione

```
yPtr = &y;
```

assegnerà al puntatore `yPtr` l'indirizzo della variabile `y`. Si dirà quindi che la variabile `yPtr` "punta a" `y`. La Figura 7.2 mostra una rappresentazione schematica della memoria in seguito all'esecuzione dell'assegnamento precedente.

La Figura 7.3 mostra la rappresentazione del puntatore all'interno della memoria, supponendo che la variabile `y` sia stata immagazzinata nella locazione `600000` e che il puntatore `yPtr` sia stato sistemato all'indirizzo `500000`. L'operando dell'operatore di indirizzo deve essere una variabile; l'operatore di indirizzo non potrà essere applicato quindi a costanti, espressioni o a variabili che siano state dichiarate con la specifica di classe di memoria `register`.

L'operatore `*`, detto comunemente *operatore di deriferimento* od *operatore di risoluzione del riferimento*, restituisce il valore dell'oggetto puntato dal suo operando (ovverosia dal puntatore). Per esempio, l'istruzione

```
printf( "%d", *yPtr );
```

visualizzerà il valore della variabile `y`, vale a dire 5. L'utilizzazione dell'operatore `*` in questo modo è detto *risoluzione del riferimento di un puntatore.*

Errore tipico 7.2

Risolvere il riferimento di un puntatore che non sia stato appropriatamente inizializzato, o che non sia stato impostato in modo da puntare a una specifica locazione di memoria. Ciò potrà causare un errore fatale in fase di esecuzione, o potrà modificare accidentalmente dei dati importanti e consentire al programma di completare la propria esecuzione, producendo però dei risultati errati.

Il programma nella Figura 7.4 mostra l'utilizzo degli operatori dei puntatori `&` e `*`. La specifica di conversione `%p` della `printf` invierà in output la locazione di memoria, convertendola in un intero esadecimale (consultate l'Appendice E, "I sistemi numerici", per ottenere maggiori informazioni sugli interi esadecimali). Noterete che nell'output l'indirizzo di `a` e il valore di `aPtr` saranno identici, confermando così che l'indirizzo di `a` sarà stato sicuramente assegnato alla variabile di tipo puntatore `aPtr` (riga 11). Gli operatori `&` e `*` sono l'uno il complemento dell'altro: qualora fossero applicati consecutivamente ad `aPtr` in qualsiasi or-

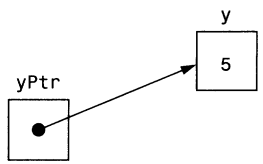


Figura 7.2 Rappresentazione grafica di un puntatore che punta a una variabile intera in memoria

Figura 7.3 Rappresentazione di y e yPtr in memoria

dine (riga 21), si otterrebbe lo stesso risultato. La tabella della Figura 7.5 mostra la priorità e l'associatività degli operatori introdotti sino a questo punto.

```
1  /* Fig. 7.4: fig07_04.c
2     Usare gli operatori & e * */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;        /* a è un intero */
8     int *aPtr;    /* aPtr è un puntatore a un intero */
9
10    a = 7;
11    aPtr = &a;    /* aPtr è impostato con l'indirizzo di a */
12
13    printf( "The address of a is %p\n"
14            "The value of aPtr is %p\n\n", &a, aPtr );
15
16    printf( "The value of a is %d\n"
17            "The value of *aPtr is %d\n\n", a, *aPtr );
18
19    printf( "Proving that * and & are complements of "
20            "each other.\n&*aPtr = %p\n*&aPtr = %p\n",
21            &*aPtr, *&aPtr );
22
23    return 0; /* indica che il programma è terminato con successo */
24
25 } /* fine della funzione main */
```

```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Proving that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

Figura 7.4 Gli operatori sui puntatori & e *

7.4 La chiamata per riferimento delle funzioni

Esistono due modi per passare gli argomenti a una funzione: per valore e per riferimento. Tutte le chiamate di funzione in C sono per valore. Come abbiamo visto nel Capitolo 5, l'istruzione `return` potrà essere utilizzata per restituire un valore da una funzione chiamata a quella chiamante (o anche per restituire il controllo da una funzione chiamata, senza restituire alcun valore). Molte funzioni però potrebbero avere la necessità di modificare alcune delle variabili ricevute dal chiamante, o di passare un puntatore a un oggetto di dati di dimensioni

Operatori	Associatività	Tipo
`( ) [ ]`	da sinistra a destra	massima
`+ - ++ -- ! * & (tipo)`	da destra a sinistra	unari
`* / %`	da sinistra a destra	moltiplicativi
`+ -`	da sinistra a destra	additivi
`< <= > >=`	da sinistra a destra	relazionali
`== !=`	da sinistra a destra	uguaglianza
`&&`	da sinistra a destra	AND logico
`||`	da sinistra a destra	OR logico
`?:`	da destra a sinistra	condizionale
`= += -= *= /= %=`	da destra a sinistra	assegnamento
`,`	da sinistra a destra	virgola

Figura 7.5 Priorità degli operatori

ragguardevoli, per evitare il dispendio causato dal passaggio del suo valore (che comporterebbe un dispendio di risorse per la creazione di una copia dell'oggetto). Per questi scopi il C fornisce la possibilità di simulare una chiamata per riferimento.

In C, i programmatori utilizzano i puntatori e l'operatore di deriferimento per simulare una chiamata per riferimento. Per richiamare una funzione con degli argomenti che debbano essere modificati, si passeranno dunque i loro indirizzi. Normalmente, ciò potrà essere ottenuto applicando l'operatore di indirizzo (`&`) alla variabile (nel chiamante) che dovrà essere modificata. I vettori invece non saranno passati utilizzando l'operatore `&` perché, come abbiamo visto nel Capitolo 6, il C passerà automaticamente la locazione di memoria del loro primo elemento (poiché il loro nome sarà sempre equivalente a `&arrayName[0]`). Nel momento in cui l'indirizzo di una variabile sarà stato passato a una funzione, nel corpo di questa potrà essere utilizzato l'operatore di deriferimento (`*`) per modificare il valore della variabile immagazzinata nella memoria del chiamante.

I programmi mostrati nella Figura 7.6 e nella Figura 7.7 corrispondono a due versioni di una funzione che eleverà al cubo un intero: `cubeByValue` e `cubeByReference`. Quello della Figura 7.6 passerà la variabile `number` alla funzione `cubeByValue` utilizzando una chiamata per valore (riga 14). La funzione `cubeByValue` eleverà al cubo il suo argomento e restituirà al `main` il nuovo valore utilizzando un'istruzione `return`. Il nuovo valore sarà assegnato a `number` nella funzione `main` (riga 14).

Il programma della Figura 7.7 passerà la variabile `number` alla funzione `cubeByReference` utilizzando una chiamata per riferimento (riga 15), ovverosia passando l'indirizzo di `number`. La funzione `cubeByReference` riceverà come parametro un puntatore a `int` chiamato `nPtr` (riga 24). La funzione risolverà il riferimento del puntatore ed eleverà al cubo il valore puntato da `nPtr` (riga 26), poi assegnerà il risultato a `*nPtr` (che corrisponde realmente a `number` nel `main`), modificando il valore di `number` nel `main`. Le Figure 7.8 e 7.9 analizzano graficamente nell'ordine i programmi della Figura 7.6 e della Figura 7.7.

Errore tipico 7.3

Non risolvere il riferimento di un puntatore, qualora sia necessario farlo per ottenere il valore cui fa riferimento il puntatore, è un errore di sintassi.

```
1  /* Fig. 7.6: fig07_06.c
2     Eleva al cubo una variabile usando una chiamata per valore */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* prototipo */
6
7  int main()
8  {
9     int number = 5; /* inizializza number */
10
11    printf( "The original value of number is %d\n", number );
12
13    /* passa number per valore a cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "The new value of number is %d\n", number );
17
18    return 0; /* indica che il programma è terminato con successo */
19
20 } /* fine della funzione main */
21
22 /* calcola e restituisce il cubo dell'argomento intero */
23 int cubeByValue( int n )
24 {
25    return n * n * n;    /* eleva al cubo la variabile locale n
26                            e restituisce il risultato */
27 } /* fine della funzione cubeByValue */
```

```
The original value of number is 5
The new value of number is 125
```

Figura 7.6 Eleva al cubo una variabile usando una chiamata per valore

```
1  /* Fig. 7.7: fig07_07.c
2     Eleva al cubo una variabile usando una chiamata per riferimento */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* prototipo */
7
8  int main()
9  {
10    int number = 5; /* inizializza number */
11
```

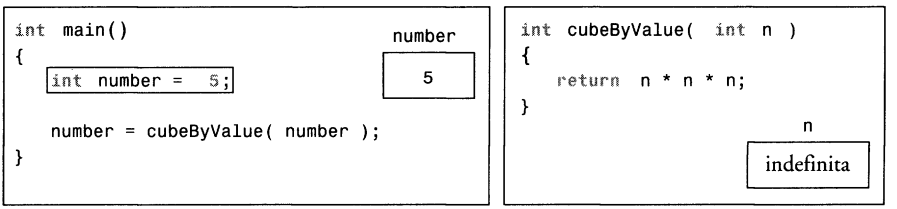
Figura 7.7 Eleva al cubo una variabile usando una chiamata per riferimento (continua)

```
12      printf( "The original value of number is %d\n", number );
13
14      /* passa l'indirizzo di number a cubeByReference */
15      cubeByReference( &number );
16
17      printf( "The new value of number is %d\n", number );
18
19      return 0; /* indica che il programma è terminato con successo */
20
21   } /* fine della funzione main */
22
23   /* calcola il cubo di *nPtr; modifica la variabile number nel main */
24   void cubeByReference( int *nPtr )
25   {
26      *nPtr = *nPtr * *nPtr * *nPtr;  /* eleva al cubo *nPtr */
27   } /* fine della funzione cubeByReference */
```

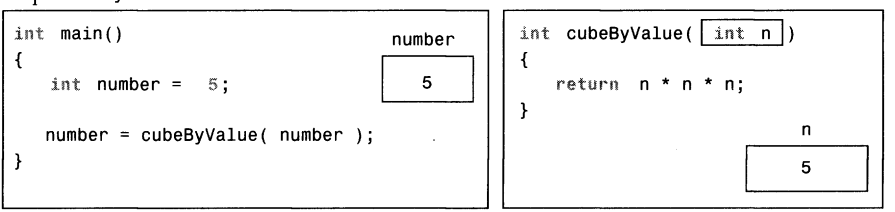
```
The original value of number is 5
The new value of number is 125
```

Figura 7.7 Eleva al cubo una variabile usando una chiamata per riferimento

All'inizio `main` richiama `cubeByValue`:



Dopo `cubeByValue` riceve la chiamata:



Dopo `cubeByValue` fa il cubo del parametro `n` e prima restituisce `cubeByValue` al `main`:

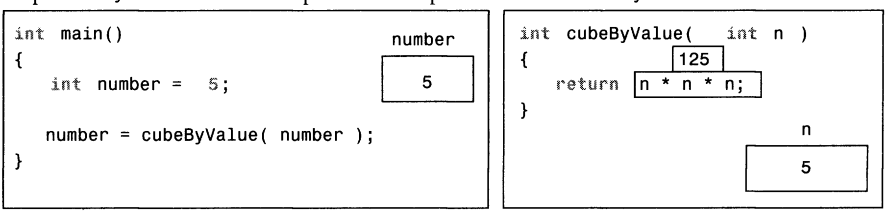


Figura 7.8 Analisi di una tipica chiamata per valore (continua)

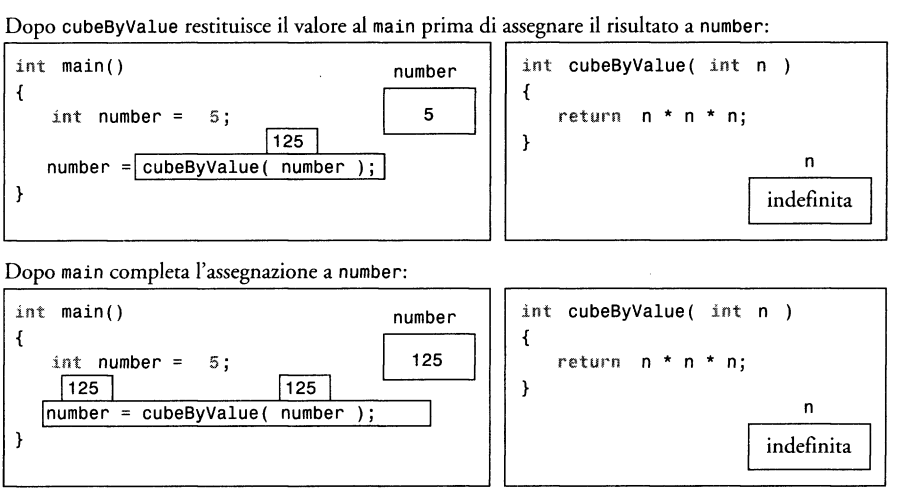


Figura 7.8 Analisi di una tipica chiamata per valore

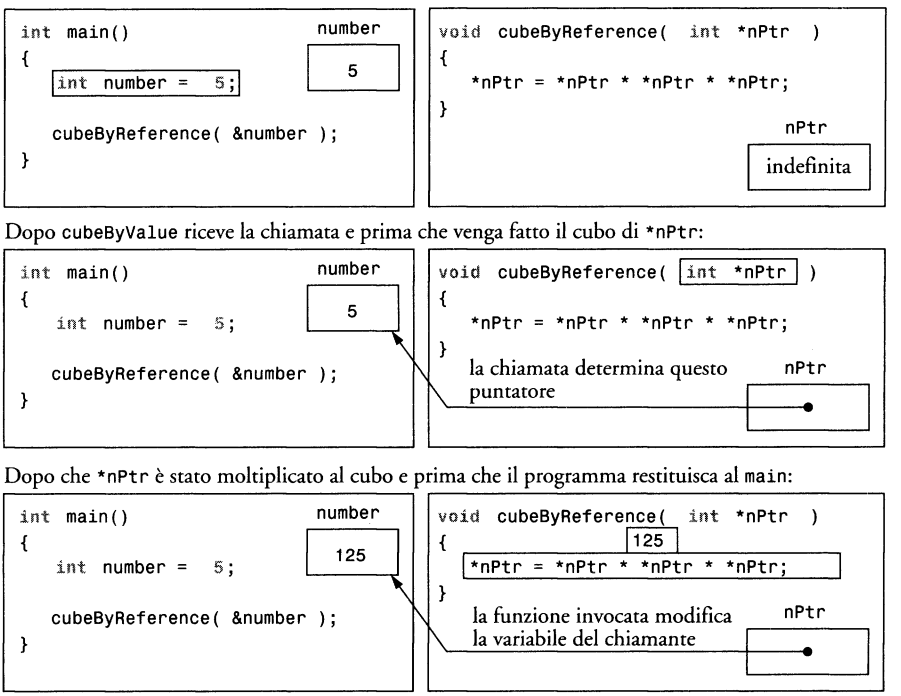


Figura 7.9 Analisi di una tipica chiamata per riferimento con un argomento puntatore

Una funzione che riceva come argomento un indirizzo dovrà definire un parametro di tipo puntatore. Per esempio, nella Figura 7.7 l'intestazione della funzione `cubeByReference` (riga 24) sarà:

```
void cubeByReference( int *nPtr )
```

La suddetta intestazione specifica che `cubeByReference` riceverà come argomento l'indirizzo di una variabile intera, immagazzinandolo localmente in `nPtr`, e che non restituirà nessun valore.

Il prototipo di funzione per `cubeByReference` conterrà `int *` all'interno delle parentesi. Così come accade con gli altri tipi di variabile, non sarà necessario includere i nomi dei puntatori all'interno dei prototipi di funzione. In ogni caso, quelli che saranno stati inclusi per scopi documentativi saranno ignorati dal compilatore C.

Qualora una funzione debba ricevere come argomento un vettore unidimensionale, nella sua intestazione e nel suo prototipo potrà essere utilizzata la stessa notazione con puntatore usata nella lista dei parametri di `cubeByReference`. Per il compilatore, una funzione che riceva un puntatore sarà equivalente a una che riceva un vettore unidimensionale. È per questo motivo che spetterà alla funzione "sapere" se stia ricevendo un vettore unidimensionale, o semplicemente una variabile per la quale sia stata utilizzata una chiamata per riferimento. Il compilatore convertirà tutti i parametri di funzione che rappresentino un vettore unidimensionale, dalla notazione `int b[]` a quella `int *b` con puntatore. Le due forme sono dunque interscambiabili.

Collaudo e messa a punto 7.2

Utilizzate sempre una chiamata per valore quando passate degli argomenti a una funzione, sempre che non sia richiesto esplicitamente che quella chiamata modifichi il valore dell'argomento nell'ambiente della funzione chiamante. Questo impedisce la modifica accidentale degli argomenti del chiamante ed è un altro esempio del principio del minimo privilegio.

7.5 Utilizzare il qualificatore const con i puntatori

Il *qualificatore const* consentirà al programmatore di informare il compilatore che il valore di una particolare variabile non dovrà essere modificato. Il qualificatore `const` non esisteva nelle prime versioni del C: è stato aggiunto al linguaggio dal comitato dell'ANSI C.

Ingegneria del software 7.1

Il qualificatore `const` *potrà essere utilizzato per applicare il principio del minimo privilegio. Applicare il principio del minimo privilegio, per progettare in modo appropriato il software, ridurrà il tempo di messa a punto e gli inopportuni effetti collaterali e renderà più facile la modifica e la manutenzione di un programma.*

Obiettivo portabilità 7.1

Per quanto il qualificatore `const` *sia ben definito nell'ANSI C, sappiate che alcuni sistemi non lo applicano.*

Nel corso degli anni, si è accumulata un'eredità formata da una gran quantità di codice scritto con le prime versioni del C, che non utilizzavano `const` perché non era ancora disponibile. Proprio per questo motivo, oggi esistono innumerevoli possibilità di miglioramento, ottenibili progettando nuovamente il software scritto con le vecchie versioni del C.

Esistono sei modi di utilizzare (o di non utilizzare) `const` con i parametri delle funzioni: due riguardano il passaggio per valore dei parametri e quattro quello per riferimento. Che

cosa guiderà la vostra scelta su una di queste sei possibilità? Lasciate che il *principio del minimo privilegio* sia la vostra guida. Assicuratevi sempre che una funzione abbia un diritto di accesso ai dati contenuti nei propri parametri tale che possa consentirgli di svolgere il suo compito, ma non di più.

Nel Capitolo 5, abbiamo spiegato che in C tutte le invocazioni di funzioni sono per valore: esse riceveranno dunque una copia dell'argomento passato attraverso l'invocazione. Il valore originale della variabile resterà immutato nell'ambiente della funzione chiamante, anche quando la sua copia sarà stata modificata all'interno della funzione chiamata. In molti casi, il valore passato a una funzione sarà modificato in modo che quella possa portare a termine il proprio compito. Esistono tuttavia delle circostanze in cui il valore di un argomento non dovrà essere alterato dalla funzione chiamata, neanche qualora si tratti di una copia del valore originale.

Considerate una funzione che riceva come argomenti un vettore unidimensionale e la sua dimensione e che visualizzi il vettore. Una tale funzione dovrà semplicemente scorrere il vettore e inviare in output ogni singolo elemento dello stesso. La sua dimensione sarà utilizzata nel corpo della funzione per determinare l'indice massimo del vettore, così che il ciclo possa determinare il momento in cui la visualizzazione sarà stata completata. Né la dimensione del vettore, né i suoi contenuti non dovrebbero dunque essere modificati nel corpo della funzione.

Collaudo e messa a punto 7.3

Nel caso in cui una variabile non sarà modificata (o non dovrà essere modificata) nel corpo della funzione cui sarà stata passata, quella variabile dovrà essere dichiarata `const` *per assicurarsi che non sia modificata accidentalmente.*

Il compilatore intercetterà qualsiasi tentativo di modificare un valore che sia stato dichiarato `const` e, secondo la particolare implementazione utilizzata, emetterà un avviso o un messaggio di errore.

Ingegneria del software 7.2

Qualora sia stata utilizzata una chiamata per valore, nell'ambiente della funzione chiamante potrà essere alterato un solo dato. Questo dovrà essere assegnato dal valore di ritorno della funzione chiamata. Per modificare più di un dato nell'ambiente di una funzione chiamante, dovrà essere utilizzata una chiamata per riferimento.

Collaudo e messa a punto 7.4

Prima di utilizzare una funzione, controllate il suo prototipo per determinare se sia in grado di modificare i valori che gli saranno passati.

Errore tipico 7.4

Ignorare che gli argomenti di una funzione invocata per riferimento siano dei puntatori, passandoli erroneamente come in una chiamata per valore. Alcuni compilatori potrebbero accettare questi valori e, dando per scontato che siano dei puntatori, risolverebbero i loro riferimenti. In fase di esecuzione, ciò provocherà spesso delle violazioni di accesso alla memoria o degli errori di segmentazione. Altri compilatori invece potrebbero rilevare la mancata corrispondenza tra i tipi degli argomenti e quelli dei parametri e genererebbero dei messaggi di errore.

Esistono ben quattro modi per passare un puntatore a una funzione: *un puntatore variabile a dati variabili*, *un puntatore costante a dati variabili*, *un puntatore variabile a dati costanti* e *un puntatore costante a dati costanti*. Ognuna delle quattro combinazioni fornisce diversi diritti di accesso. Questi ultimi saranno discussi nei prossimi esempi.

Convertire in caratteri maiuscoli una stringa usando un puntatore variabile a dati variabili

Il livello più alto di accesso ai dati sarà garantito da un puntatore variabile a dati variabili. In questo caso, ovviamente dopo averne risolto il riferimento, i dati potranno essere modificati attraverso il puntatore e questo potrà essere cambiato in modo che possa fare riferimento ad altri dati. La dichiarazione di un puntatore variabile a dati variabili non includerà il qualificatore `const`. Un puntatore di questo genere potrebbe essere utilizzato, per esempio, per ricevere un argomento di tipo stringa in una funzione che utilizzi *l'aritmetica dei puntatori* per elaborare (ed eventualmente modificare) i caratteri della stringa. La funzione `convertToUppercase` della Figura 7.10 dichiarerà come suo parametro proprio un puntatore variabile a dati variabili chiamandolo `sPtr` (`char *sPtr`) in riga 23. La funzione elaborerà la stringa `string` (puntata da `sPtr`) un carattere per volta usando l'aritmetica dei puntatori. La funzione di libreria standard del C `islower` (chiamata in riga 27) controlla il carattere contenuto nell'indirizzo puntato da `sPtr`. Nel caso che un carattere sia compreso nell'intervallo `a`–`z`, `islower` restituirà vero e la funzione della libreria standard del C `toupper` (riga 28) verrà invocata per convertire il carattere nella sua corrispondente lettera maiuscola; in caso contrario, `islower` restituirà falso e sarà elaborato il carattere successivo della stringa.

```
1  /* Fig. 7.10: fig07_10.c
2     Convertire le lettere minuscole in lettere maiuscole
3     usando un puntatore variabile a dati variabili */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototipo */
9
10 int main()
11 {
12    char string[] = "characters and $32.98"; /* inizializza il vettore
                                                  di caratteri */
13
14    printf( "The string before conversion is: %s\n", string );
15    convertToUppercase( string );
16    printf( "The string after conversion is:  %s\n", string );
17
18    return 0; /* indica che il programma è terminato con successo */
19
20 } /* fine della funzione main */
21
22 /* converte la stringa in caratteri maiuscoli */
```

Figura 7.10 Convertire una stringa in maiuscolo usando un puntatore variabile a dati variabili (continua)

```
23  void convertToUppercase( char *sPtr )
24  {
25     while ( *sPtr != '\0' ) { /* il carattere corrente non è '\0' */
26
27        if ( islower( *sPtr ) ) { /* se il carattere è minuscolo */
28           *sPtr = toupper( *sPtr ); /* lo converte in maiuscolo */
29        } /* fine del comando if */
30
31        ++sPtr;  /* incrementa sPtr per puntare al prossimo carattere */
32     } /* fine del comando while */
33
34  } /* fine della funzione convertToUppercase */
```

```
The string before conversion is: characters and $32.98
The string after conversion is:  CHARACTERS AND $32.98
```

Figura 7.10 Convertire una stringa in maiuscolo usando un puntatore variabile a dati variabili

Visualizzare una stringa un carattere per volta usando un puntatore variabile a dati costanti

Un puntatore variabile a dati costanti potrà essere modificato in modo che possa fare riferimento ad altri dati, purché siano del tipo appropriato, ma questi non potranno essere modificati. Un puntatore di questo genere potrà essere utilizzato, per esempio, per ricevere un argomento di tipo vettore in una funzione che elabori ogni elemento di quest'ultimo, senza modificarne i dati. Per esempio, la funzione `printCharacters` della Figura 7.11 dichiarerà il parametro `sPtr` di tipo `const char *` (riga 24). La dichiarazione va letta da destra a sinistra come "`sPtr` è un puntatore a una costante di carattere". Il corpo della funzione utilizzerà un comando `for` per inviare in output i caratteri della stringa, finche non sarà stato incontrato quello nullo. Una volta che ogni carattere sarà stato stampato, il puntatore `sPtr` sarà incrementato così che possa puntare al carattere successivo della stringa.

```
1  /* Fig. 7.11: fig07_11.c
2     Visualizzare una stringa un carattere per volta
3     usando un puntatore variabile a dati costanti */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main()
10 {
11    /* inizializza il vettore di caratteri */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
```

Figura 7.11 Visualizzare una stringa un carattere per volta usando un puntatore variabile a dati costanti (continua)

```
15      printCharacters( string );
16      printf( "\n" );
17
18      return 0; /* indica che il programma è terminato con successo */
19
20   } /* fine della funzione main */
21
22   /* sPtr non può modificare il carattere a cui punta,
23      ovvero, è un puntatore di "sola lettura" */
24   void printCharacters( const char *sPtr )
25   {
26      /* itera scorrendo l'intera stringa */
27      for ( ; *sPtr != '\0'; sPtr++ ) { /* non c'è inizializzazione */
28         printf( "%c", *sPtr );
29      } /* fine del comando for */
30
31   } /* fine della funzione printCharacters */
```

```
The string is:
print characters of a string
```

Figura 7.11 Visualizzare una stringa un carattere per volta usando un puntatore variabile a dati costanti

La Figura 7.12 mostra i messaggi di errore prodotti tentando di compilare una funzione che riceva un puntatore variabile (`xPtr`) a dati costanti. Questa funzione tenterà di modificare i dati puntati da `xPtr` in riga 22 producendo un errore di compilazione. [*Nota*: l'effettivo messaggio d'errore che vedrete dipende dal compilatore.]

```
1   /* Fig. 7.12: fig07_12.c
2      Tentare di modificare i dati
3      per mezzo di un puntatore variabile a dati costanti */
4   #include <stdio.h>
5
6   void f( const int *xPtr ); /* prototipo */
7
8   int main()
9   {
10     int y; /* dichiara y */
11
12     f( &y ); /* f tenta di eseguire una modifica illecita */
13
14     return 0; /* indica che il programma è terminato con successo */
15
16  } /* fine della funzione main */
17
```

Figura 7.12 Tentare di modificare i dati per mezzo di un puntatore variabile a dati costanti (continua)

```
18   /* xPtr non può essere usato per modificare
19      il valore della variabile a cui punta */
20   void f( const int *xPtr )
21   {
22      *xPtr = 100; /* errore: non si può modificare un oggetto const */
23   } /* fine della funzione f */
```

```
Compiling...
FIG07_12.C
d:\books\2003\chtp4\examples\ch7\fig07_12.c(22) : error C2166: l-value
     specifies const object
Error executing cl.exe

FIG07_12.exe - 1 error(s), 0 warning(s)
```

Figura 7.12 Tentare di modificare i dati per mezzo di un puntatore variabile a dati costanti

I vettori, come sappiamo, sono un tipo di dato aggregato che immagazzinano, sotto un unico nome, informazioni correlate e dello stesso tipo. Nel Capitolo 10, discuteremo un altro tipo di dato aggregato detto *struttura* (ma anche *record* in altri linguaggi). Una struttura è in grado di immagazzinare, sotto un unico nome, molte informazioni correlate e di tipo differente (per esempio, potrà immagazzinare i dati di ogni impiegato di un'azienda). Un vettore sarà passato automaticamente per riferimento nell'invocazione di una funzione che lo riceva come argomento. Le strutture invece saranno sempre passate per valore: sarà dunque passata una copia dell'intera struttura. In fase di esecuzione, ciò richiederà un certo dispendio di tempo e di spazio causato dalla preparazione di una copia di tutti gli elementi presenti nella struttura e, ovviamente, dal relativo immagazzinamento nella memoria delle funzioni chiamate. Potremo invece utilizzare dei puntatori a dati costanti, qualora dovessimo fornire a una funzione le informazioni contenute in una struttura, in modo da combinare l'efficienza di una chiamata per riferimento con la protezione di una chiamata per valore. Nel passaggio di un puntatore a una struttura, dovrà essere preparata solo la copia dell'indirizzo in cui quella sarà stata immagazzinata. Su una macchina che utilizzi degli indirizzi formati da 4 byte, saranno ricopiati solo questi pochi byte di memoria, invece delle centinaia o forse delle migliaia di byte che la struttura potrebbe includere.

Obiettivo efficienza 7.1

Passate gli oggetti di dimensioni ragguardevoli, come le strutture, utilizzando dei puntatori a dati costanti in modo da combinare i benefici dell'efficienza di una chiamata per riferimento con la sicurezza di una chiamata per valore.

Un siffatto utilizzo dei puntatori a dati costanti è un esempio di *compromesso tempo/spazio*. Infatti, sarà preferibile utilizzare i puntatori nei casi in cui la memoria sia scarsa e l'efficienza sia un interesse di primaria importanza. Mentre, nei casi in cui la memoria sia abbondante e l'efficienza non sia un interesse di primaria importanza, sarà preferibile passare i dati per valore, in modo da rispettare il principio del minimo privilegio. Ricordate che alcuni sistemi non implementano correttamente il qualificatore `const`, perciò la chiamata per valore sarà sempre il modo migliore per prevenire la modifica dei dati.

Tentare di modificare un puntatore costante a dati variabili

Un puntatore costante a dati variabili fa riferimento sempre alla stessa posizione di memoria, mentre i dati che vi sono immagazzinati potranno essere modificati per mezzo dello stesso. Questo tipo di puntatore è il default per il nome di un vettore. Questo, infatti, è un puntatore costante all'inizio del vettore. Tutti i dati del vettore potranno essere letti e modificati utilizzando il suo nome e un indice. Un puntatore costante a dati variabili potrà essere utilizzato, per esempio, per ricevere un argomento di tipo vettore in una funzione che acceda ai suoi elementi utilizzando esclusivamente la notazione con gli indici di vettore. I puntatori dichiarati con il qualificatore `const` dovranno essere inizializzati contestualmente alla loro dichiarazione (qualora il puntatore sia il parametro di una funzione, sarà inizializzato con quello che sarà stato passato nella sua invocazione). Il programma della Figura 7.13 tenterà di modificare un puntatore costante, `ptr` che sarà stato dichiarato in riga 12 di tipo `int * const`. La dichiarazione va letta da destra a sinistra come "`ptr` è un puntatore costante a un intero". Il puntatore sarà inizializzato (riga 12) con l'indirizzo della variabile intera `x` e, quando il programma tenterà di assegnare a `ptr` l'indirizzo di `y` (riga 15), sarà generato un messaggio di errore.

```
1  /* Fig. 7.13: fig07_13.c
2     Tentare di modificare un puntatore costante a dati variabili */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x; /* dichiara x */
8     int y; /* dichiara y */
9
10    /* ptr è un puntatore costante a un intero che può essere
11       modificato per mezzo di ptr, ma ptr punta sempre alla stessa
         locazione di memoria */
12    int * const ptr = &x;
13
14    *ptr = 7; /* permesso: *ptr non è const */
15    ptr = &y; /* errore: ptr è const; non gli si può assegnare
                   un nuovo indirizzo */
16
17    return 0; /* indica che il programma è terminato con successo */
18
19 } /* fine della funzione main */
```

```
Compiling...
FIG07_13.C
d:\books\2003\chtp4\examples\ch7\FIG07_13.c(15) : error C2166: l-value
     specifies const object
Error executing cl.exe

FIG07_13.exe - 1 error(s), 0 warning(s)
```

Figura 7.13 Tentare di modificare un puntatore costante a dati variabili

Tentare di modificare un puntatore costante a dati costanti

Il diritto di accesso minimo sarà garantito da un puntatore costante a dati costanti. Un puntatore di questo genere farà riferimento sempre alla stessa locazione di memoria e i dati che vi saranno contenuti non potranno essere modificati. Questo è il tipo di puntatore che dovrebbe essere utilizzato, per esempio, per passare un vettore a una funzione che la veda utilizzando solo la notazione con gli indici di vettore e che non modifichi i valori dei suoi elementi. Il programma della Figura 7.14 dichiarerà un puntatore `ptr` (riga 13) di tipo `const int * const`, dichiarazione che va letta da destra a sinistra come "`ptr` è un puntatore costante a una costante intera". La figura mostra i messaggi di errore che saranno generati, qualora si tenti di modificare i dati puntati da `ptr` (riga 17) o l'indirizzo immagazzinato nel puntatore (riga 18).

```
1  /* Fig. 7.14: fig07_14.c
2     Tentare di modificare un puntatore costante a dati costanti */
3  #include <stdio.h>
4
5  int main()
6  {
7     int x = 5; /* inizializza x */
8     int y; /* dichiara y */
9
10    /* ptr è un puntatore costante a una costante intera,
11       ptr punta sempre alla stessa locazione;
12       l'intero in quella locazione non può essere modificato */
13    const int * const ptr = &x;
14
15    printf( "%d\n", *ptr );
16
17    *ptr = 7; /* errore: *ptr è const: non gli si può assegnare
                   un nuovo valore */
18    ptr = &y; /* errore: ptr è const; non gli si può assegnare
                   un nuovo indirizzo */
19
20    return 0; /* indica che il programma è terminato con successo */
21
22 } /* fine della funzione main */
```

```
Compiling...
FIG07_14.C
d:\books\2003\chtp4\examples\ch7\FIG07_14.c(17) : error C2166: l-value
     specifies const object
d:\books\2003\chtp4\examples\ch7\FIG07_14.c(18) : error C2166: l-value
     specifies const object
Error executing cl.exe

FIG07_14.exe - 2 error(s), 0 warning(s)
```

Figura 7.14 Tentare di modificare un puntatore costante a dati costanti

7.6 L'ordinamento a bolle utilizzando una chiamata per riferimento

Modificheremo ora il programma per l'ordinamento a bolle della Figura 6.15, in modo da utilizzare due funzioni: `bubbleSort` e `swap`. La funzione `bubbleSort` eseguirà l'ordinamento del vettore e richiamerà `swap` (riga 53), per scambiare di posto gli elementi `array[ j ]` e `array[ j + 1 ]` del vettore (consultate la Figura 7.15). Ricordate che il C applica l'incapsulamento delle informazioni tra le funzioni, perciò `swap` non avrà accesso ai singoli elementi del vettore di `bubbleSort`. Dato che questa *desidera* che `swap` abbia accesso agli elementi del vettore così che possa scambiarli di posto, `bubbleSort` passerà a `swap` ognuno di quegli elementi attraverso una chiamata per riferimento: in altri termini, sarà passato in modo esplicito l'indirizzo di ogni elemento del vettore. Un vettore completo, come sappiamo, sarebbe passato automaticamente per riferimento, ma i singoli elementi di un vettore sono degli scalari e saranno passati normalmente per valore. Di conseguenza, nella chiamata a `swap` (riga 53), `bubbleSort` utilizzerà l'operatore di indirizzo (&) con ognuno degli elementi del vettore, nel modo seguente

```
swap( &array[ j ], &array[ j + 1 ] );
```

per eseguire una chiamata per riferimento. La funzione `swap` riceverà `&array[ j ]` nel puntatore `element1Ptr` (riga 64). Sebbene alla funzione `swap` non sia consentito conoscere il nome `array[ j ]` a causa dell'incapsulamento delle informazioni, essa potrà comunque utilizzare `*element1Ptr` come un sinonimo per `array[ j ]`. Di conseguenza, quando `swap` farà riferimento a `*element1Ptr`, in realtà starà puntando ad `array[ j ]` di `bubbleSort`. In modo simile, quando `swap` farà riferimento a `*element2Ptr`, in realtà starà puntando ad `array[ j + 1 ]` di `bubbleSort`. Per quanto a `swap` non siano consentite delle istruzioni come

```
hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

lo stesso preciso effetto sarà ottenuto inserendo le istruzioni dalla riga 66 alla riga 68

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

nella funzione `swap` della Figura 7.15.

```
 1  /* Fig. 7.15: fig07_15.c
 2     Questo programma memorizza valori in un vettore, li ordina
 3     in modo ascendente e visualizza il vettore risultante */
 4  #include <stdio.h>
 5  #define SIZE 10
 6
 7  void bubbleSort( int * const array, const int size ); /* prototipo */
 8
 9  int main()
10  {
11     /* inizializza a */
```

Figura 7.15 Bubble sort con chiamata per riferimento (continua)

```
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    int i; /* contatore */
15
16    printf( "Data items in original order\n" );
17
18    /* itera scorrendo il vettore a */
19    for ( i = 0; i < SIZE; i++ ) {
20       printf( "%4d", a[ i ] );
21    } /* fine del comando for */
22
23    bubbleSort( a, SIZE); /* ordina il vettore */
24
25    printf( "\nData items in ascending order\n" );
26
27    /* itera scorrendo il vettore a */
28    for ( i = 0; i < SIZE; i++ ) {
29       printf( "%4d", a[ i ] );
30    } /* fine del comando for */
31
32    printf( "\n" );
33
34    return 0; /* indica che il programma è terminato con successo */
35
36 } /* fine della funzione main */
37
38 /* ordina un vettore di interi usando l'algoritmo bubble sort */
39 void bubbleSort( int * const array, const int size )
40 {
41    void swap( int *element1Ptr, int *element2Ptr ); /* prototipo */
42    int pass; /* contatore dei passaggi */
43    int j; /* contatore dei confronti */
44
45    /* ciclo per controllare i passaggi */
46    for ( pass = 1; pass < size - 1; pass++ ) {
47
48       /* ciclo per controllare i confronti per ogni passaggio */
49       for ( j = 0; j < size - 1; j++ ) {
50
51          /* scambia gli elementi adiacenti se non sono in ordine */
52          if ( array[ j ] > array[ j + 1 ] ) {
53             swap( &array[ j ], &array[ j + 1 ] );
54          } /* fine del comando if */
55
56       } /* fine del comando for interno */
57
58    } /* fine del comando for esterno */
59
60 } /* fine della funzione bubbleSort */
```

Figura 7.15 Bubble sort con chiamata per riferimento (continua)

```
61
62    /* scambia i valori nelle locazioni di memoria
63       puntate da element1Ptr e element2Ptr */
64    void swap( int *element1Ptr, int *element2Ptr )
65    {
66       int hold = *element1Ptr;
67       *element1Ptr = *element2Ptr;
68       *element2Ptr = hold;
69    } /* fine della funzione swap */
```

```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Figura 7.15 Bubble sort con chiamata per riferimento

Molte caratteristiche della funzione `bubbleSort` sono davvero degne di nota. L'intestazione della funzione (riga 39) dichiarerà `array` come `int *array`, invece che come `int array[]`, per indicare che `bubbleSort` riceverà come argomento un vettore unidimensionale (ripetiamo ancora una volta che queste notazioni sono interscambiabili). Il parametro `size` sarà dichiarato `const` per obbedire al principio del minimo privilegio. Per quanto il parametro `size` riceva una copia del valore conservato nella funzione `main` e per quanto la sua modifica non potrà alterare il dato originario, `bubbleSort` non avrà bisogno di variare `size` per eseguire il suo compito. Infatti, la dimensione del vettore resterà fissa durante l'esecuzione di `bubbleSort`. Di conseguenza, `size` sarà dichiarata `const` per assicurarsi che non sia modificata accidentalmente. È probabile che l'algoritmo di ordinamento non funzioni correttamente, qualora la dimensione del vettore sia modificata durante l'elaborazione.

Il prototipo della funzione `swap` (riga 41) è stato incluso nel corpo di `bubbleSort` poiché questa sarà l'unica funzione che la richiamerà. Inserire il prototipo in `bubbleSort` limiterà le invocazioni accettabili della funzione `swap` esclusivamente a quelle eseguite all'interno di `bubbleSort`. Le altre funzioni che dovessero tentare di invocare `swap` non avrebbero accesso al giusto prototipo di funzione, perciò spetterebbe al compilatore generarne uno automaticamente. Ma ciò produrrà normalmente un prototipo che non corrisponderà all'intestazione della funzione e genererà un errore del compilatore, poiché questo presumerà che il valore restituito dalla funzione e i suoi parametri siano di tipo `int`.

Ingegneria del software 7.3

Inserire i prototipi all'interno delle definizioni delle altre funzioni obbedisce al principio del minimo privilegio, attraverso la limitazione delle invocazioni ammissibili alle funzioni in cui compaiono i prototipi.

Osservate che la funzione `bubbleSort` riceverà come parametro la dimensione del vettore (riga 39). Per poterlo ordinare, infatti, la funzione dovrà conoscere la sua dimensione. Nel passaggio di un vettore a una funzione, questa riceve l'indirizzo di memoria relativo al suo primo elemento. Naturalmente l'indirizzo non fornisce però alcuna informazione riguardante il numero degli elementi presenti nel vettore. Di conseguenza, toccherà al programmatore fornire alla funzione la dimensione del vettore.

Nel programma, la dimensione del vettore sarà passata alla funzione `bubbleSort` in modo esplicito. Questo approccio offre due principali benefici: la riusabilità del software e una corretta progettazione dello stesso. Definendo la funzione in modo tale che riceva la dimensione del vettore per mezzo di un argomento, faremo sì che la funzione possa essere utilizzata da un qualsiasi programma che abbia la necessità di ordinare dei vettori unidimensionali di interi di qualsiasi dimensione.

Ingegneria del software 7.4

Quando passate un vettore a una funzione, fornite anche la sua dimensione. Ciò aiuterà a rendere la funzione riutilizzabile in molti programmi.

Avremmo potuto immagazzinare la dimensione del vettore in una variabile globale che fosse accessibile all'intero programma. Questo approccio sarebbe stato anche più efficiente, poiché non ci sarebbe stata la necessità di creare una copia della dimensione durante l'invocazione della funzione. Tuttavia, gli altri programmi che eventualmente richiedessero l'ordinamento di vettori di interi potrebbero non contenere la stessa variabile globale e, di conseguenza, non potrebbero utilizzare la nostra funzione.

Ingegneria del software 7.5

Le variabili globali violano il principio del minimo privilegio e sono un esempio di una scadente progettazione del software.

Obiettivo efficienza 7.2

Il passaggio della dimensione di un vettore a una funzione richiederà del tempo e avrà bisogno di uno spazio aggiuntivo nell'ambiente della funzione, poiché dovrà essere preparata una copia della dimensione. Le variabili globali, invece, non richiederanno tempo e spazio aggiuntivi, perché vi si potrà accedere direttamente da ogni funzione.

Avremmo anche potuto inserire la dimensione del vettore direttamente nella funzione. Questo approccio però avrebbe limitato l'utilizzo della funzione a vettori di una particolare dimensione e avrebbe ridotto notevolmente la sua riusabilità. La funzione, infatti, sarebbe stata utilizzabile solo da programmi che avessero elaborato vettori unidimensionali di interi con una dimensione uguale a quella codificata all'interno della funzione.

7.7 L'operatore sizeof

Il C fornisce l'apposito operatore unario *sizeof* per determinare la dimensione in byte di un vettore, o di ogni altro tipo di dato, durante la fase di compilazione. Utilizzato con il nome di un vettore, come mostrato nella Figura 7.16 (riga 14), l'operatore `sizeof` restituirà un valore intero pari al numero totale di byte contenuto nello stesso. Osservate che le variabili di tipo `float` saranno normalmente immagazzinate in 4 byte di memoria e che `array` sarà dichiarato con 20 elementi. Di conseguenza, in `array` ci saranno in totale 80 byte.

Obiettivo efficienza 7.3

`sizeof` *è un operatore applicabile a tempo di compilazione; quindi non comporta nessun dispendio di risorse al momento dell'esecuzione.*

```
1   /* Fig. 7.16: fig07_16.c
2      Quando viene applicato al nome di un vettore,
3     l'operatore sizeof restituisce il numero dei byte di quest'ultimo */
4   #include <stdio.h>
5
6   size_t getSize( float *ptr ); /* prototipo */
7
8   int main()
9   {
10     float array[ 20 ]; /* crea il vettore */
11
12     printf( "The number of bytes in the array is %d"
13                "\nThe number of bytes returned by getSize is %d\n",
14               sizeof( array ), getSize( array ) );
15
16     return 0; /* indica che il programma è terminato con successo */
17
18  } /* fine della funzione main */
19
20  /* restituisce la dimensione di ptr */
21  size_t getSize( float *ptr )
22  {
23     return sizeof( ptr );
24
25  } /* fine della funzione getSize */
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Figura 7.16 Quando viene applicato al nome di un vettore, l'operatore sizeof restituisce il numero dei byte di quest'ultimo

Anche il numero degli elementi di un vettore potrà essere determinato per mezzo dell'operatore `sizeof`. Considerate, per esempio, la seguente dichiarazione di vettore:

```
double real[ 22 ];
```

Le variabili di tipo `double` saranno normalmente immagazzinate in 8 byte di memoria. Di conseguenza, il vettore `real` conterrà un totale di 176 byte. Per determinare il numero degli elementi presenti nel vettore, potrà essere utilizzata la seguente espressione:

```
sizeof( real ) / sizeof( double )
```

Questa determinerà il numero di byte contenuti nel vettore `real` e dividerà quel valore per la quantità di byte utilizzata per immagazzinare un valore `double` in memoria.

Notate che la funzione `getSize` ha come tipo di ritorno `size_t`. Il tipo `size_t` è un tipo definito dallo standard C come il tipo intero (`unsigned` o `unsigned long`) restituito dall'operatore `sizeof`. Il tipo `size_t` è definito nel file di intestazione `<stddef.h>` (che viene incluso da molti altri file di intestazione, come ad esempio `<stdio.h>`). Il programma della Figura 7.17 calcolerà il numero di byte utilizzati per immagazzinare i tipi di dato standard. I risultati possono differire da un computer all'altro.

```
1   /* Fig. 7.17: fig07_17.c
2      Dimostrazione dell'uso dell'operatore sizeof */
3   #include <stdio.h>
4
5   int main()
6   {
7      char c;
8      short s;
9      int i;
10     long l;
11     float f;
12     double d;
13     long double ld;
14     int array[ 20 ]; /* crea un vettore di 20 elementi di tipo int */
15     int *ptr = array; /* crea un puntatore al vettore */
16
17     printf( "     sizeof c = %d\tsizeof(char)  = %d"
18             "\n   sizeof s = %d\tsizeof(short) = %d"
19             "\n   sizeof i = %d\tsizeof(int) = %d"
20             "\n   sizeof l = %d\tsizeof(long) = %d"
21             "\n   sizeof f = %d\tsizeof(float) = %d"
22             "\n   sizeof d = %d\tsizeof(double) = %d"
23             "\n   sizeof ld = %d\tsizeof(long double) = %d"
24             "\n sizeof array = %d"
25             "\n   sizeof ptr = %d\n",
26           sizeof c, sizeof ( char ), sizeof s, sizeof ( short ),
                sizeof i,
27           sizeof ( int ), sizeof l, sizeof ( long ), sizeof f,
28          sizeof ( float ), sizeof d, sizeof ( double ), sizeof ld,
29           sizeof ( long double ), sizeof array, sizeof ptr );
30
31     return 0; /* indica che il programma è terminato con successo */
32
33  } /* fine della funzione main */
```

```
sizeof c = 1          sizeof(char)  = 1
sizeof s = 2          sizeof(short) = 2
sizeof i = 4          sizeof(int) = 4
sizeof l = 4          sizeof(long) = 4
sizeof f = 4          sizeof(float) = 4
sizeof d = 8      sizeof(double) = 8
sizeof ld = 8     sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Figura 7.17 Usare l'operatore sizeof per determinare le dimensioni dei tipi di dato standard

Obiettivo portabilità 7.2

Il numero di byte utilizzati per immagazzinare un particolare tipo di dato potrebbe variare tra i diversi sistemi. Nello scrivere dei programmi che dipendano dalle dimensioni dei tipi di dato e che debbano essere eseguiti su diversi sistemi di computer, utilizzate `sizeof` *per determinare il numero di byte utilizzato per immagazzinare i vari tipi di dato.*

L'operatore `sizeof` potrà essere utilizzato con ogni nome di variabile, tipo di dato o valore (incluso il valore di un'espressione). Nel caso che sia utilizzato con il nome di una variabile che non sia un vettore, o con quello di una costante, sarà restituito il numero di byte usati per immagazzinare il tipo specifico di quella variabile o costante. Osservate che le parentesi utilizzate con `sizeof` saranno obbligatorie, qualora il suo operando corrisponda al nome di un tipo di dato. Le parentesi non saranno necessarie, invece, qualora il suo operando corrisponda al nome di una variabile.

7.8 Le espressioni con i puntatori e l'aritmetica dei puntatori

I puntatori sono dei validi operandi per le espressioni aritmetiche, quelle di assegnamento e quelle di confronto. Tuttavia, non tutti gli operatori utilizzati normalmente in questi tipi di espressione saranno ammissibili in combinazione con le variabili di tipo puntatore. Questa sezione descriverà proprio quegli operatori che potranno accettare dei puntatori come loro operandi e del modo in cui questi operatori saranno utilizzati.

Con gli operatori potrà essere utilizzato un insieme limitato di operazioni aritmetiche. Un puntatore potrà essere *incrementato* (++) o *decrementato* (--), vi si potrà *sommare* (+ o +=) o *sottrarre* (- o -=) *un intero*, o si potrà *sottrarre un puntatore da un altro*.

Supponete che il vettore `int v[ 5 ]` sia stato dichiarato e che il suo primo elemento si trovi nella locazione di memoria `3000`. Supponete anche che il puntatore `vPtr` sia stato inizializzato in modo da fare riferimento a `v[ 0 ]`: in altri termini, che il valore di `vPtr` sia `3000`. La Figura 7.18 rappresenta con un diagramma proprio questa situazione, in una macchina in cui gli interi siano di 4 byte. Osservate che `vPtr` potrà essere inizializzato, per far sì che punti al vettore `v`, con una delle seguenti istruzioni:

```
vPtr = v;
vPtr = &v[ 0 ];
```

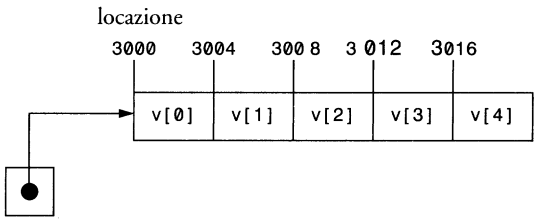


Figura 7.18 Un vettore v e una variabile puntatore vPtr che punta a v

Obiettivo portabilità 7.3

La maggior parte dei computer utilizzano oggigiorno interi di 2 o 4 byte. Alcune delle macchine più recenti utilizzano invece degli interi di 8 byte. L'aritmetica dei puntatori è dipendente dalla macchina, giacché i suoi risultati dipendono dalla dimensione degli oggetti puntati.

Nell'aritmetica convenzionale, la somma `3000 + 2` restituisce il valore `3002`. Questo però non è normalmente il caso dell'aritmetica dei puntatori. Nel momento in cui si sommerà o si sottrarrà un intero da un puntatore, questo non sarà semplicemente incrementato o decrementato con quel valore, bensì di quell'intero moltiplicato per la dimensione dell'oggetto cui il puntatore farà riferimento. La dimensione dell'oggetto dipenderà dal suo tipo di dato. Per esempio, l'istruzione

```
vPtr += 2;
```

produrrà `3008` (`3000 + 2 * 4`), supponendo che un intero sia immagazzinato in 4 byte di memoria. Nel vettore `v`, `vPtr` punterebbe ora all'elemento `v[ 2 ]` (Figura 7.19). Nel caso che un intero sia immagazzinato in 2 byte di memoria, allora il calcolo precedente avrebbe prodotto come risultato la locazione di memoria `3004` (`3000 + 2 * 2`). Qualora il vettore fosse stato di un tipo di dato differente, il calcolo precedente avrebbe incrementato il puntatore di due volte il numero di byte necessari per immagazzinare un oggetto di quel tipo di dato. I risultati dell'aritmetica dei puntatori combaceranno con gli esiti di quella regolare, solo quando saranno eseguite delle operazioni sui vettori di caratteri, poiché ognuno di essi è lungo 1 byte.

Qualora `vPtr` fosse stato incrementato a `3016`, che punterebbe a `v[ 4 ]`, l'istruzione

```
vPtr -= 4;
```

avrebbe impostato `vPtr` di nuovo a `3000`, ovverosia all'inizio del vettore. Qualora un puntatore debba essere incrementato o decrementato di uno, potranno essere utilizzati gli operatori di incremento (`++`) e di decremento (`--`). Entrambe le seguenti istruzioni

```
++vPtr;
vPtr++;
```

incrementeranno il puntatore in modo da fargli fare riferimento alla posizione successiva del vettore. Entrambe le seguenti istruzioni

```
--vPtr;
vPtr--;
```

decrementeranno il puntatore in modo da fargli fare riferimento all'elemento precedente del vettore.

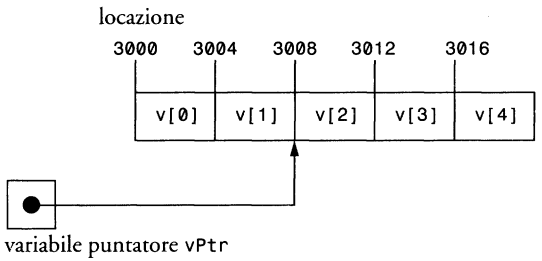


Figura 7.19 Il puntatore vPtr dopo l'operazione di aritmetica dei puntatori

Le variabili di tipo puntatore possono essere sottratte l'una dall'altra. Per esempio, se `vPtr` contenesse la locazione `3000` e `v2Ptr` contenesse l'indirizzo `3008`, l'istruzione

```
x = v2Ptr - vPtr;
```

assegnerebbe a `x` il numero degli elementi del vettore compresi tra `vPtr` e `v2Ptr`, in questo caso `2` (non 8). L'aritmetica dei puntatori non avrebbe senso qualora non fosse utilizzata con un vettore. Non possiamo presumere che due variabili dello stesso tipo siano immagazzinate in modo contiguo nella memoria, sempre che quelle non siano due elementi adiacenti di un vettore.

Errore tipico 7.5

Utilizzare l'aritmetica dei puntatori su un puntatore che non faccia riferimento a un elemento di un vettore.

Errore tipico 7.6

Sottrarre o confrontare due puntatori che non facciano riferimento a elementi dello stesso vettore.

Errore tipico 7.7

Superare uno dei due limiti di un vettore quando si utilizza l'aritmetica dei puntatori.

Un puntatore potrà essere assegnato a un altro puntatore, qualora siano entrambi dello stesso tipo. L'eccezione a questa regola è il puntatore a `void` (ovverosia `void *`), perché questo è generico e può rappresentare qualsiasi tipo di puntatore. Un puntatore a `void` potrà essere assegnato a tutti gli altri tipi di puntatori e questi potranno essere assegnati a un puntatore a `void`. In entrambi i casi non sarà necessaria alcuna operazione di conversione.

Non è possibile risolvere il riferimento di un puntatore a `void`. Per esempio, il compilatore sa che un puntatore a `int` punta a quattro byte di memoria, su una macchina con interi di 4 byte, ma un puntatore a `void` contiene semplicemente una locazione di memoria per un tipo di dato sconosciuto: di conseguenza, il compilatore non è in grado di conoscere il numero preciso dei byte cui il puntatore fa riferimento. Il compilatore deve dunque conoscere il tipo di dato, per determinare il numero di byte in cui si risolverà il riferimento di uno specifico puntatore.

Errore tipico 7.8

Assegnare un puntatore di un tipo a uno di un altro tipo qualora nessuno dei due sia un `void *` *è un errore di sintassi.*

Errore tipico 7.9

Tentare di risolvere il riferimento di un puntatore `void *` *è un errore di sintassi.*

I puntatori potranno essere confrontati utilizzando gli operatori di uguaglianza e quelli relazionali, ma tali confronti non avranno significato qualora i puntatori non facciano riferimento a membri dello stesso vettore. I confronti tra puntatori raffrontano gli indirizzi che vi sono immagazzinati. Un confronto tra due puntatori che si riferiscono allo stesso vettore potrebbe mostrare, per esempio, che un puntatore fa riferimento a un elemento con un

indice di vettore più alto di quello puntato dall'altro. Un utilizzo tipico dei confronti tra puntatori è di determinare se un puntatore sia NULL.

7.9 La relazione tra i puntatori e i vettori

I vettori e i puntatori sono strettamente correlati in C e potranno essere utilizzati in modo *quasi* interscambiabile. Il nome di un vettore potrebbe essere considerato come un puntatore costante. I puntatori potranno essere utilizzati per svolgere qualsiasi operazione che coinvolga gli indici di un vettore.

Si assume che il vettore di interi `b[ 5 ]` e la variabile di tipo puntatore a interi `bPtr` siano già stati dichiarati. Dato che il nome del vettore (senza l'indice) è un puntatore al suo primo elemento, potremo impostare `bPtr` con l'indirizzo del primo elemento del vettore `b` con l'istruzione:

```
bPtr = b;
```

Questa istruzione è equivalente all'indirizzo del primo elemento del vettore rilevato nel modo seguente:

```
bPtr = &b[ 0 ];
```

L'elemento del vettore `b[ 3 ]` potrà anche essere puntato usando l'espressione con puntatore:

```
*( bPtr + 3 )
```

Il `3` nella suddetta espressione è l'*offset* (lo scostamento) dal puntatore. L'offset indicherà quale elemento del vettore dovrà essere puntato e coinciderà con il suo indice, qualora il puntatore faccia riferimento all'inizio di un vettore. La notazione precedente è detta *notazione con puntatore e offset*. Le parentesi sono necessarie perché la priorità di `*` è maggiore di quella di `+`. Senza le parentesi, la suddetta espressione avrebbe aggiunto `3` al valore di `*bPtr` (in altre parole, il `3` sarebbe stato aggiunto a `b[ 0 ]`, sempre supponendo che `bPtr` punti all'inizio del vettore). L'elemento di un vettore potrà dunque essere puntato con una espressione di tipo puntatore, ma anche il suo indirizzo

```
&b[ 3 ]
```

potrà essere ottenuto con l'espressione di tipo puntatore

```
bPtr + 3
```

Lo stesso vettore potrà essere trattato come un puntatore e utilizzato sfruttando le regole della relativa aritmetica. Per esempio, anche l'espressione

```
*( b + 3 )
```

punterà all'elemento `b[ 3 ]` del vettore. In generale, tutte le espressioni con gli indici di vettore potranno essere convertite in quelle con puntatore e offset. In questo caso specifico, abbiamo utilizzato una notazione con puntatore e offset in cui il nome del vettore è stato utilizzato nella sua veste di puntatore. Osservate che l'istruzione precedente non modificherà in alcun modo il nome del vettore: `b` punterà ancora al suo primo elemento.

I puntatori potranno essere usati in combinazioni con gli indici esattamente come i vettori. Per esempio, nel caso in cui `bPtr` assuma come valore `b`, l'espressione

```
bPtr[ 1 ]
```

punterà all'elemento `b[ 1 ]` del vettore. Questa è appunto la *notazione con puntatore e indice.*

Ricordate che il nome di un vettore è essenzialmente un puntatore costante: esso fa sempre riferimento all'inizio del vettore. Di conseguenza, l'espressione

```
b += 3
```

non sarà valida, perché tenterà di modificare il valore del nome del vettore con l'aritmetica dei puntatori.

Errore tipico 7.10

Tentare di modificare il nome di un vettore con l'aritmetica dei puntatori è un errore di sintassi.

Il programma della Figura 7.20 utilizzerà i quattro metodi per puntare agli elementi di un vettore di cui abbiamo discusso ("con gli indici di vettore", "con puntatore e offset" usando il nome del vettore come puntatore, "con gli indici di puntatore", "con puntatore e offset" usando un puntatore) per visualizzare i quattro elementi del vettore di interi `b`.

Per illustrare ulteriormente l'interscambiabilità dei vettori e dei puntatori, daremo uno sguardo alle due funzioni per la copia di stringhe, `copy1` e `copy2`, incluse nel programma della Figura 7.21. Entrambe le funzioni copieranno una stringa (ovverosia un vettore di caratteri) in un altro vettore di caratteri. Dopo aver confrontato i prototipi per `copy1` e `copy2` le due funzioni ci sembreranno identiche. Infatti, eseguiranno lo stesso compito, ma nonostante le apparenze, le due funzioni sono state implementate in maniera differente.

La funzione `copy1` utilizzerà la notazione con gli indici di vettore per copiare la stringa `s2` nel vettore di caratteri `s1`. La funzione dichiarerà un contatore, la variabile intera `i`, che però utilizzerà solo come indice del vettore. L'intera operazione di copia sarà eseguita dall'intestazione del comando `for` (riga 31): infatti, il suo corpo è un'istruzione vuota. L'intestazione specifica che `i` sarà azzerata e incrementata di uno a ogni iterazione del ciclo. La condizione del comando `for`, `s1[ i ] = s2[ i ]`, eseguirà l'operazione di copia da `s2` a `s1` un carattere per volta. Nel momento in cui sarà stato incontrato il carattere nullo di `s2`, esso sarà assegnato a `s1` e il valore dell'assegnamento diventerà il valore assegnato all'operando sinistro (`s1`). Il ciclo terminerà, poiché il valore intero del carattere nullo è zero (falso).

```
1  /* Fig. 7.20: fig07_20.c
2     Usare le notazioni con indici e puntatori per i vettori */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; /* inizializza il vettore b */
9     int *bPtr = b;   /* fa puntare bPtr al vettore b */
10    int i; /* contatore */
11    int offset; /* contatore */
12
13    /* visualizza il vettore b usando la notazione con gli indici
         di vettore */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
```

Figura 7.20 Usare i quattro metodi di puntamento agli elementi di un vettore (continua)

```
16      /* itera scorrendo il vettore b */
17      for ( i = 0; i < 4; i++ ) {
18         printf( "b[ %d ] = %d\n", i, b[ i ] );
19      } /* fine del comando for */
20
21      /* visualizza il vettore b usando il suo nome e la notazione
           con puntatore e offset */
22      printf( "\nPointer/offset notation where \n"
23              "the pointer is the array name\n" );
24
25      /* itera scorrendo il vettore b */
26      for ( offset = 0; offset < 4; offset++ ) {
27         printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28      } /* fine del comando for */
29
30      /* visualizza il vettore b usando bPtr e la notazione
           con gli indici di vettore */
31      printf( "\nPointer subscript notation\n" );
32
33      /* itera scorrendo il vettore b */
34      for ( i = 0; i < 4; i++ ) {
35         printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ]);
36      } /* fine del comando for */
37
38      /* visualizza il vettore b usando bPtr e la notazione
           con puntatore e offset */
39      printf( "\nPointer/offset notation\n" );
40
41      /* itera scorrendo il vettore b */
42      for ( offset = 0; offset < 4; offset++ )
43         printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44      } /* fine del comando for */
45
46      return 0; /* indica che il programma è terminato con successo */
47
48   } /* fine della funzione main */
```

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40
```

Figura 7.20 Usare i quattro metodi di puntamento agli elementi di un vettore (continua)

```
Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Figura 7.20 Usare i quattro metodi di puntamento agli elementi di un vettore

La funzione `copy2` utilizzerà invece i puntatori e la loro aritmetica per copiare la stringa `s2` nel vettore di caratteri `s1`. Anche in questo caso l'intestazione del comando `for` (riga 41) eseguirà l'intera operazione di copia. L'intestazione non include nessuna inizializzazione di variabili. Proprio come nella funzione `copy1`, la condizione (`*s1 = *s2`) eseguirà l'operazione di copia. Dopo aver risolto i riferimenti dei due puntatori, il carattere puntato da `s2` sarà assegnato all'elemento cui `s1` farà riferimento. Dopo l'esecuzione dell'assegnamento nella condizione, i puntatori saranno entrambi incrementati in modo da puntare rispettivamente all'elemento successivo del vettore `s1` e al prossimo carattere della stringa `s2`. Nel momento in cui sarà stato incontrato il carattere nullo della stringa `s2`, questo sarà assegnato all'elemento puntato da `s1` e il ciclo terminerà.

Osservate che il primo argomento di entrambe le funzioni, `copy1` e `copy2`, dovrà essere un vettore sufficientemente grande per contenere la stringa del secondo argomento. In caso contrario potrà verificarsi un errore, qualora si tentasse di scrivere in una locazione di memoria che non appartenga al vettore. Osservate anche che il secondo parametro di ognuna delle due funzioni è dichiarato come `const char *` (una stringa costante). Questo perché in entrambe le funzioni il secondo argomento sarà ricopiato nel primo: i suoi caratteri saranno letti uno per volta, ma non saranno mai modificati. Per questo motivo, il secondo parametro è stato dichiarato in modo da puntare a un valore costante, rispettando così il principio del minimo privilegio. Nessuna delle due funzioni ha la necessità di modificare il secondo argomento, perciò tale possibilità non è stata concessa a nessuna delle due.

```
1   /* Fig. 7.21: fig07_21.c
2       Copiare una stringa usando la notazione dei vettori e quella
        dei puntatori */
3   #include <stdio.h>
4
5   void copy1( char *s1, const char *s2 ); /* prototipo */
6   void copy2( char *s1, const char *s2 ); /* prototipo */
7
8   int main()
9   {
```

Figura 7.21 Copiare una stringa usando la notazione dei vettori e quella dei puntatori (continua)

```
10      char string1 = [ 10 ]; /* crea il vettore string1 */
11      char *string2 = "Hello"; /* crea un puntatore a una stringa */
12      char string3 = [ 10 ]; /* crea il vettore string3 */
13     char string4[] = "Good Bye"; /* crea un puntatore a una stringa */
14
15      copy1( string1, string2 );
16      printf( "string1 = %s\n", string1 );
17
18      copy2( string3, string4 );
19      printf( "string3 = %s\n", string3 );
20
21      return 0; /* indica che il programma è terminato con successo */
22
23   } /* fine della funzione main */
24
25   /* copia s2 in s1 usando la notazione dei vettori */
26   void copy1( char *s1, const char *s2 )
27   {
28      int i; /* contatore */
29
30      /* itera scorrendo le stringhe */
31      for( i = 0; ( s1 [ i ] = s2[ i ] ) != '\0'; i++) {
32         ; /* non fa nulla nel corpo */
33      } /* fine del comando for */
34
35   } /* fine della funzione copy1 */
36
37   /* copia s2 in s1 usando la notazione dei puntatori */
38   void copy2( char *s1, const char *s2 )
39   {
40      /* itera scorrendo le stringhe */
41      for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42         ; /* non fa nulla nel corpo */
43      } /* fine del comando for */
44
45   } /* fine della funzione copy2 */
```

```
string1 = Hello
string3 = Good Bye
```

Figura 7.21 Copiare una stringa usando la notazione dei vettori e quella dei puntatori

7.10 I vettori di puntatori

I vettori possono contenere anche dei puntatori. Un utilizzo tipico per un vettore di puntatori è la formazione di un *vettore di stringhe*. Ogni elemento del vettore è una stringa, ma in C questa è essenzialmente un puntatore al suo primo carattere. Di conseguenza, ogni elemento incluso in un vettore di stringhe sarà in realtà un puntatore al primo carattere di una stringa. Considerate ora la dichiarazione di un vettore di stringhe `suit` che possa essere utilizzato per rappresentare un mazzo di carte da gioco:

```
char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

La porzione `suit[ 4 ]` della dichiarazione indica un vettore di 4 elementi. La porzione `char *` della dichiarazione indica che ogni elemento del vettore `suit` sarà del tipo "puntatore a `char`". Il qualificatore `const` indica che le stringhe puntate da ogni puntatore elemento del vettore non saranno modificate. I quattro valori che dovranno essere inseriti nel vettore sono `"Hearts"` (Cuori), `"Diamonds"` (Quadri), `"Clubs"` (Fiori) e `"Spades"` (Picche). Ognuno di questi sarà immagazzinato in memoria come una stringa di caratteri terminata da un `NULL` e, proprio per questo, tale stringa sarà lunga un carattere in più di quelli compresi tra le virgolette. Dunque le quattro stringhe saranno lunghe rispettivamente 7, 9, 6 e 7 caratteri. Per quanto le apparenze possano lasciarvi supporre che queste stringhe saranno sistemate nel vettore `suit`, in realtà in questo saranno immagazzinati soltanto dei puntatori (Figura 7.22). Ogni puntatore punterà al primo carattere della stringa corrispondente. Di conseguenza, sebbene la dimensione del vettore `suit` sia fissa, questo ci permetterà di accedere a stringhe di qualsiasi lunghezza. Questa flessibilità è solo un esempio delle potenti possibilità per la strutturazione dei dati del linguaggio C.

Avremmo potuto inserire i semi delle carte in una matrice, sistemando in ognuna delle righe un seme e in ogni colonna le singole lettere dei loro nomi. Una tale struttura di dati, però, avrebbe dovuto avere un numero fisso di colonne pari alla lunghezza della stringa più lunga. Ne conseguirebbe uno spreco di memoria causato dagli elementi che contengono le stringhe più corte: uno spreco che diventerebbe molto più consistente, qualora immagazzinassimo un gran numero di stringhe e molte di queste fossero più corte di quella più lunga. Nella prossima sezione utilizzeremo i vettori di stringhe per rappresentare un mazzo di carte.

7.11 Studio di un caso: simulazione di un mescolatore e distributore di carte

In questa sezione, utilizzeremo la generazione dei numeri casuali per sviluppare un programma che simuli un mescolatore e distributore di carte da gioco. In seguito, questo programma potrà essere utilizzato per svilupparne altri che implementino dei giochi di carte specifici. Per evidenziare alcuni impercettibili problemi di efficienza, abbiamo utilizzato intenzionalmente degli algoritmi di mescolamento e distribuzione non ottimizzati. Negli esercizi e nel Capitolo 10, svilupperemo degli algoritmi più efficienti.

Utilizzando l'approccio *top-down per raffinamenti successivi*, svilupperemo un programma che mescolerà e distribuirà un mazzo di 52 carte da gioco. L'approccio top-down è particolarmente utile quando si affrontano problemi più corposi e complessi di quelli che abbiamo affrontato nei capitoli precedenti.

Utilizziamo la matrice 4 per 13 `deck` per rappresentare il mazzo delle carte da gioco (Figura 7.23). Le righe corrisponderanno ai semi: la riga 0 corrisponderà ai cuori, la 1 ai

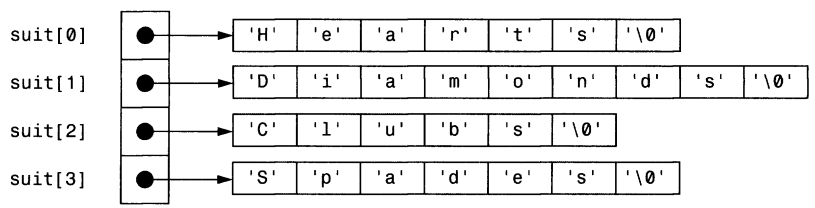


Figura 7.22 Un esempio grafico della matrice suit

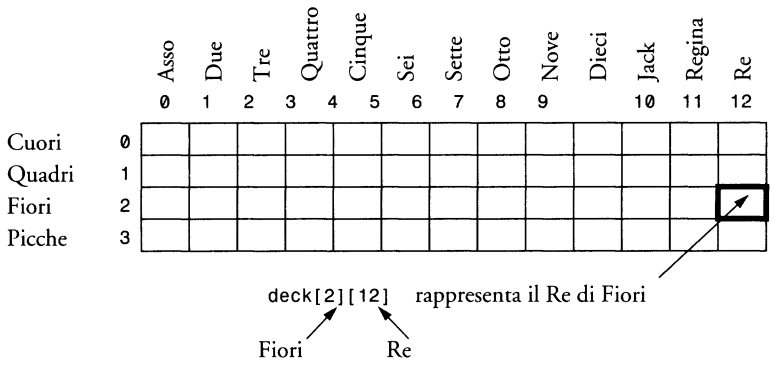


Figura 7.23 Rappresentazione della matrice utilizzata per il mazzo di carte

quadri, la 2 ai fiori e la 3 alle picche. Le colonne corrisponderanno invece ai valori delle carte: quelle da 0 a 9 corrisponderanno rispettivamente ai valori dall'asso al dieci, mentre quelle dal 10 al 12 corrisponderanno al fante, alla regina e al re. Dovremo caricare i vettori `suit` e `face` con le stringhe di caratteri che rappresentano rispettivamente i quattro semi e i tredici valori delle carte.

Il mazzo di carte simulato potrà essere mischiato come segue. La matrice `deck` sarà in primo luogo azzerata. In seguito, saranno scelte a caso una `row` (riga da 0 a 3) e una `column` (colonna da 0 a 12). Nell'elemento della matrice `deck[row][column]` sarà inserito il numero 1, per indicare che questa sarà la prima carta che sarà distribuita dal mazzo mescolato. Il processo continuerà con l'inserimento dei numeri 2, 3, ... 52 in posizioni della matrice `deck` scelte a caso, per indicare quale carta sarà sistemata nel mazzo mescolato come seconda, terza, ... cinquantaduesima. Man mano che la matrice `deck` comincerà a riempirsi, diventerà sempre più probabile l'estrazione di una carta già selezionata (`deck[row][column]` diversa da zero). Questo genere di selezioni dovrà semplicemente essere ignorato e si dovranno eseguire altre scelte casuali di `row` e `column`, finché non sarà stata ritrovata una carta che non sia già stata selezionata. Alla fine i numeri da 1 a 52 occuperanno le 52 caselle della matrice `deck`. Solo a questo punto il mazzo di carte sarà stato mescolato completamente.

Questo algoritmo di mescolamento potrebbe girare all'infinito, qualora la selezione casuale continuasse a proporre delle carte che siano già state inserite tra quelle mescolate. Questo fenomeno è noto come *differimento indefinito*. Negli esercizi discuteremo un algoritmo di mescolamento migliore che eliminerà la possibilità di un differimento indefinito.

Obiettivo efficienza 7.4

A volte un algoritmo che venga in mente in modo "spontaneo" potrebbe contenere degli impercettibili problemi di efficienza come un differimento indefinito. Ricercate degli algoritmi che evitino il differimento indefinito.

Per distribuire la prima carta, dovremo ritrovare all'interno della matrice l'elemento che soddisfi la condizione `deck[ row ][ column ] = 1`. La ricerca sarà eseguita con un comando `for` nidificato che varierà `row` da 0 a 3 e `column` da 0 a 12. A quale carta corrisponderà quell'elemento della matrice? Il vettore `suit` sarà stato precaricato con i quattro semi, perciò per ottenere quello della carta appena individuata visualizzeremo la stringa di caratteri `suit[ row ]`. In modo simile, per ottenere il valore di quella carta stamperemo la stringa di caratteri

`face[ column ]`. Stamperemo anche la stringa di caratteri `" of "` (di), combinandola appropriatamente con le altre informazioni in modo da visualizzare ogni carta nella forma `"King of Clubs"` (Re di Fiori), `"Ace of Diamonds"` (Asso di Quadri) e così via.

Procediamo ora con il processo top–down per raffinamenti successivi. Il top sarà semplicemente

Mischiare e distribuire 52 carte

Il nostro primo raffinamento produrrà:

Inizializzare il vettore dei semi
Inizializzare il vettore dei valori
Inizializzare la matrice del mazzo di carte
Mischiare il mazzo di carte
Distribuire le 52 carte

"Mischiare il mazzo di carte" potrà essere espanso come segue:

Per ognuna delle 52 carte
    Inserire il numero di estrazione in una casella libera del mazzo di carte scelta a caso

"Distribuire le 52 carte" potrà essere espanso come segue:

Per ognuna delle 52 carte
    Trovare il numero di estrazione nella matrice del mazzo di carte e visualizzare il valore e il seme della carta

Incorporando le suddette espansioni otterremo il nostro secondo passo di raffinamento completo:

Inizializzare il vettore dei semi
Inizializzare il vettore dei valori
Inizializzare la matrice del mazzo di carte

Per ognuna delle 52 carte
    Inserire il numero di estrazione in una casella libera del mazzo di carte scelta a caso

Per ognuna delle 52 carte
    Trovare il numero di estrazione nella matrice del mazzo di carte e visualizzare il valore e il seme della carta

"Inserire il numero di estrazione in una casella libera del mazzo di carte scelta a caso" potrà essere espansa in:

Scegliere a caso una casella del mazzo di carte

Finché la casella del mazzo di carte è già stata scelta in precedenza
    Scegliere a caso una casella del mazzo di carte

Inserire il numero di estrazione nella casella del mazzo di carte scelta

"Trovare il numero di estrazione nella matrice del mazzo di carte e visualizzare il valore e il seme della carta" potrà essere espanso come segue:

Per ogni casella della matrice del mazzo di carte
    Se la casella contiene il numero di estrazione
        Visualizzare il valore e il seme della carta

Incorporando queste espansioni otterremo il nostro terzo passo di raffinamento:

Inizializzare il vettore dei semi
Inizializzare il vettore dei valori
Inizializzare la matrice del mazzo di carte

Per ognuna delle 52 carte
  Scegliere a caso una casella del mazzo di carte

Finché la casella del mazzo di carte è già stata scelta in precedenza
  Scegliere a caso una casella del mazzo di carte

Inserire il numero di estrazione nella casella del mazzo di carte scelta

Per ognuna delle 52 carte
  Per ogni casella della matrice del mazzo di carte
    Se la casella contiene il numero di estrazione
      Visualizzare il valore e il seme della carta

E con questo passo avremo completato il processo di raffinamento. Osservate che questo programma potrebbe essere più efficiente, qualora le porzioni dell'algoritmo che si occupano del mescolamento e della distribuzione fossero combinate, in modo da distribuire le carte nello stesso momento in cui saranno state sistemate nel mazzo. Abbiamo scelto di implementare separatamente queste operazioni perché le carte normalmente sono distribuite dopo essere state mescolate (non mentre si mescolano).

Il programma di mescolamento e distribuzione delle carte è mostrato della Figura 7.24 mentre nella Figura 7.25 è stato riportato un esempio di esecuzione. Osservate l'utilizzo della specifica di conversione `%s` nell'invocazione di `printf` per visualizzare le stringhe di caratteri. L'argomento corrispondente a quella specifica nella chiamata di `printf` dovrà essere un puntatore a `char` (o un vettore `char`). Nella funzione `deal`, la specifica di formato `"%5s of %-8s"` (riga 76) visualizzerà una stringa allineata a destra in un campo di cinque caratteri seguita da `" of "` e da una stringa allineata a sinistra in un campo di otto caratteri. Il segno meno in `%-8s` significa che la stringa sarà allineata a sinistra in un campo di lunghezza 8.

```
1   /* Fig. 7.24: fig07_24.c
2      Programma per mescolare e distribuire delle carte da gioco */
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <time.h>
6
7   /* prototipi */
8   void shuffle( int wDeck[][ 13 ] );
9   void deal( const int wDeck[][ 13 ], const char *wFace[],
10                    const char *wSuit[] );
11
12  int main()
13  {
14     /* inizializza il vettore dei semi */
15     const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
16
17     /* inizializza il vettore delle facce */
```

Figura 7.24 Programma per la distribuzione delle carte da gioco (continua)

```
18    const char *face[ 13 ] =
19                    { "Ace", "Deuce", "Three", "Four",
20                      "Five", "Six", "Seven", "Eight",
21                      "Nine", "Ten", "Jack", "Queen", "King" };
22
23    /* inizializza il vettore del mazzo di carte */
24    int deck[ 4 ][ 13 ] = { 0 };
25
26    srand( time( NULL ) ); /* specifica il seme per il generatore
                                di numeri casuali */
27
28    shuffle( deck );
29    deal( deck, face, suit );
30
31    return 0; /* indica che il programma è terminato con successo */
32
33 } /* fine della funzione main */
34
35 /* mescola le carte nel mazzo */
36 void shuffle( int wDeck[][ 13 ] )
37 {
38    int row; /* numero di riga */
39    int column; /* numero di colonna */
40    int card; /* contatore */
41
42    /* per ognuna delle 52 carte, sceglie a caso una casella
         del mazzo */
43    for( card = 1; card <= 52; card++ ) {
44
45       /* sceglie a caso una nuova locazione finché non trova
            una casella libera */
46       do {
47          row = rand() % 4;
48          column = rand() % 13;
49       } while(wDeck[ row ][ column ] != 0); /* fine del comando
                                                   do...while */
50
51       /* memorizza il numero della carta nella casella del mazzo
            scelta */
52       wDeck[ row ][ column ] = card;
53    } /* fine del comando for */
54
55 } /* fine della funzione shuffle */
56
57 /* distribuisce le carte nel mazzo */
58 void deal( const int wDeck[][ 13 ], const char *wFace[],
59            const char *wSuit[] )
60 {
61    int card; /* contatore delle carte */
62    int row; /* contatore delle righe */
63    int column; /* contatore delle colonne */
```

Figura 7.24 Programma per la distribuzione delle carte da gioco (continua)

```
64
65       /* distribuisce ognuna delle 52 carte */
66       for ( card = 1; card <= 52; card++ ) {
67
68          /* itera scorrendo le righe di wDeck */
69          for ( row = 0; row <= 3; row++ ) {
70
71             /* itera scorrendo le colonne di wDeck relative alla riga
                corrente */
72             for ( column = 0; column <= 12; column++ ) {
73
74                /* se la casella contiene la carta corrente,
                   la visualizza */
75                if ( wDeck[ row ][ column ] == card) {
76                   printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
77                           card % 2 == 0 ? '\n' : '\t' );
78                } /* fine del comando if */
79
80             } /* fine del comando for */
81
82          } /* fine del comando for */
83
84       } /* fine del comando for */
85
86    } /* fine della funzione deal */
```

Figura 7.24 Programma per la distribuzione delle carte da gioco

```
Nine of Hearts          Five of Clubs
Queen of Spades         Three of Spades
Queen of Hearts            Ace of Clubs
 King of Hearts            Six of Spades
 Jack of Diamonds         Five of Spades
Seven of Hearts           King of Clubs
Three of Clubs          Height of Hearts
Three of Diamonds         Four of Diamonds
Queen of Diamonds         Five of Diamonds
   Six of Diamonds        Five of Hearts
   Ace of Spades             Six of Hearts
 Nine of Diamonds       Queen of Clubs
Eight of Spades           Nine of Clubs
Deuce of Clubs               Six of Clubs
Deuce of Spades           Jack of Clubs
 Four of Clubs          Eight of Clubs
 Four of Spades         Seven of Spades
Seven of Diamonds       Seven of Clubs
 King of Spades             Ten of Diamonds
 Jack of Hearts             Ace of Hearts
```

Figura 7.25 Esempio di esecuzione del programma per la distribuzione delle carte da gioco (continua)

```
 Jack of Spades          Ten of Clubs
Eight of Diamonds      Deuce of Diamonds
  Ace of Diamonds       Nine of Spades
 Four of Hearts        Deuce of Hearts
 King of Diamonds         Ten of Spades
Three of Hearts           Ten of Hearts
```

Figura 7.25 Esempio di esecuzione del programma per la distribuzione delle carte da gioco

C'è un punto debole nel nostro algoritmo di distribuzione. I due comandi `for` più interne continueranno a cercare una corrispondenza tra gli elementi di `deck`, anche quando quella sarà già stata trovata e magari al primo tentativo. Correggeremo questo difetto negli esercizi e nello studio di un caso del Capitolo 10.

7.12 I puntatori a funzioni

Un *puntatore a una funzione* contiene il suo indirizzo di memoria. Nel Capitolo 6 abbiamo visto che il nome di un vettore è in realtà l'indirizzo di memoria del suo primo elemento. In modo simile, il nome di una funzione è in realtà l'indirizzo di memoria dal quale inizia il codice che eseguirà il compito della funzione. Questo tipo di puntatori potrà essere passato e restituito dalle funzioni, immagazzinato in vettori e assegnato ad altri puntatori a funzioni.

Per mostrare l'utilizzo dei puntatori a funzioni la Figura 7.26 presenta una versione modificata del programma per l'ordinamento a bolle della Figura 7.15. La nuova versione è formata da un `main` e dalle funzioni `bubble`, `swap`, `ascending` e `descending`. La funzione `bubble` riceverà come argomento un puntatore a una funzione, `ascending` o `descending`, oltre che un vettore di interi e la sua dimensione. Il programma richiederà all'utente di scegliere se il vettore debba essere ordinato in modo ascendente o discendente. Qualora l'utente immetta 1, alla funzione `bubble` sarà passato un puntatore ad `ascending`, determinando così l'ordinamento ascendente del vettore. Qualora l'utente immetta 2, alla funzione `bubble` sarà passato un puntatore a `descending`, determinando così l'ordinamento discendente del vettore. L'output del programma è mostrato nella Figura 7.27.

```
1  /* Fig. 7.26: fig07_26.c
2     Programma di ordinamento multifunzione che fa uso dei puntatori
      a funzioni */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* prototipi */
7  void bubble( int work[], const int size, int (*compare)( int a,
     int b ) );
8  int ascending(int a, int b);
9  int descending(int a, int b);
10
```

Figura 7.26 Programma di ordinamento multifunzione che fa uso dei puntatori a funzioni (continua)

```
11  int main()
12  {
13     int order; /* 1 per ordinamento ascendente o 2 per ordinamento
                     discendente */
14     int counter; /* contatore */
15
16     /* inizializza il vettore a */
17     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19     printf( "Enter 1 to sort in ascending order,\n"
20              "Enter 2 to sort in descending order: " );
21     scanf( "%d", &order );
22
23     printf( "\nData items in original order\n" );
24
25     /* visualizza il vettore originale */
26     for ( counter = 0; counter < SIZE; counter++ ) {
27        printf( "%5d", a[ counter ] );
28     } /* fine del comando for */
29
30     /* ordina il vettore in ordine ascendente; passa la funzione
          ascending
31        come argomento per specificare l'ordinamento ascendente */
32     if ( order == 1 ) {
33        bubble( a, SIZE, ascending );
34        printf( "\nData items in ascending order\n" );
35     } /* fine del ramo if */
36     else { /* passa la funzione descending */
37        bubble( a, SIZE, descending );
38        printf( "\nData items in descending order\n" );
39     } /* fine del ramo else */
40
41     /* visualizza il vettore ordinato */
42     for ( counter = 0; counter < SIZE; counter++ ) {
43        printf( "%5d", a[ counter ] );
44     } /* fine del comando for */
45
46     printf( "\n" );
47
48     return 0; /* indica che il programma è terminato con successo */
49
50  } /* fine della funzione main */
51
52  /* bubble sort multifunzione;  il parametro compare è un puntatore
53     a una funzione di comparazione che determina il senso
          dell'ordinamento */
54  void bubble( int work[], const int size, int (*compare)( int a,
       int b ) )
55  {
```

Figura 7.26 Programma di ordinamento multifunzione che fa uso dei puntatori a funzioni (continua)

```
56       int pass; /* contatore dei passaggi */
57       int count; /* contatore dei confronti */
58
59       void swap( int *element1Ptr, int *element2Ptr ); /* prototipo */
60
61       /* ciclo per controllare i passaggi */
62       for ( pass = 1; pass < size; pass++ ) {
63
64          /* ciclo per controllare i confronti per ogni passaggio */
65          for ( count = 0; count < size - 1; count++ ) {
66
67             /* scambia gli elementi adiacenti se non sono in ordine */
68             if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69                swap( &work[ count ], &work[ count + 1 ] );
70             } /* fine del comando if */
71
72          } /* fine del comando for */
73
74       } /* fine del comando for */
75
76    } /* fine della funzione bubble */
77
78    /* scambia i valori nelle locazioni di memoria
79       puntate da element1Ptr e element2Ptr */
80    void swap( int *element1Ptr, int *element2Ptr )
81    {
82       int hold; /* variabile temporanea */
83
84       int hold = *element1Ptr;
85       *element1Ptr = *element2Ptr;
86       *element2Ptr = hold;
87    } /* fine della funzione swap */
88
89    /* determina se gli elementi non sono in ordine
90       rispetto a un ordinamento ascendente */
91    int ascending( int a, int b )
92    {
93       return b < a; /* effettua uno scambio se b è minore di a */
94
95    } /* fine della funzione ascending */
96
97    /* determina se gli elementi non sono in ordine
98       rispetto a un ordinamento discendente */
99    int descending( int a, int b )
100   {
101      return b > a; /* effettua uno scambio se b è maggiore di a */
102
103   } /* fine della funzione descending */
```

Figura 7.26 Programma di ordinamento multifunzione che fa uso dei puntatori a funzioni

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in ascending order
    2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2    6    4    8   10   12   89   68   45   37
Data items in descending order
   89   68   45   37   12   10    8    6    4    2
```

Figura 7.27 I risultati del programma di ordinamento a bolle della Figura 7.26

Nell'intestazione della funzione `bubble` (riga 54) appare il seguente parametro:

```
int (*compare) ( int a, int b )
```

Questo avverte `bubble` che riceverà un parametro (`compare`) che è un puntatore a una funzione che accetta due argomenti interi e che restituisce un risultato dello stesso tipo. Le parentesi intorno a `*compare` sono necessarie per raggruppare `*` con `compare` in modo da indicare che quest'ultimo è un puntatore. Se non avessimo incluso le parentesi, la dichiarazione sarebbe diventata

```
int *compare ( int a, int b )
```

che avrebbe dichiarato una funzione che riceve due parametri interi e restituisce un puntatore dello stesso tipo.

Il prototipo della funzione `bubble` è mostrato in riga 7. Notate che il prototipo avrebbe potuto essere scritto come

```
int (*) ( int, int )
```

senza il nome del puntatore a funzione e i nomi dei parametri.

La funzione passata a `bubble` sarà invocata all'interno di un'istruzione `if` (riga 68) come segue

```
if ( (*compare)( work[ count ], work[ count + 1 ] ) )
```

Occorre risolvere il riferimento di un puntatore a una funzione per invocarla, proprio come è necessario risolvere il riferimento di un puntatore a una variabile per accedere al suo contenuto.

Avremmo potuto invocare la funzione senza risolvere il riferimento del puntatore, come nell'istruzione

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

che utilizzerebbe direttamente il puntatore in sostituzione del nome della funzione. Abbiamo però preferito il primo metodo di invocazione, perché mostra in modo esplicito che `compare`

è un puntatore di cui è stato risolto il riferimento per richiamare la funzione. Nel secondo metodo di invocazione il puntatore `compare` sembra essere invece il nome di una funzione. Ovviamente, ciò potrebbe creare confusione in un utente che desiderasse dare uno sguardo alla definizione della funzione `compare` e scoprisse che non è mai stata definita nel file.

Usare i puntatori a funzioni per creare un sistema guidato da menu

I puntatori a funzioni sono utilizzati tipicamente nei cosiddetti sistemi guidati da menu. In questi sistemi si richiede all'utente di selezionare un'opzione da un menu (per esempio, da 1 a 5). Ogni opzione sarà servita da una funzione differente. I puntatori che fanno riferimento a ognuna di quelle funzioni sono immagazzinati in un vettore di puntatori a funzione. La scelta dell'utente sarà utilizzata come indice del vettore per rilevare il puntatore che sarà utilizzato per invocare la funzione.

Il programma della Figura 7.28 fornisce un esempio generico dei meccanismi per la dichiarazione e l'utilizzo di un vettore di puntatori a funzioni. Sono state definite tre funzioni, `function1`, `function2` e `function3`, ognuna delle quali riceverà un argomento intero e non restituirà alcun valore. I puntatori che fanno riferimento a queste tre funzioni sono stati immagazzinati nel vettore `f`, dichiarato (riga 14) come segue:

```
void (*f[ 3 ]) ( int ) = { function1, function2, function3 };
```

La dichiarazione va letta cominciando dall'insieme di parentesi più a sinistra, "`f` è una matrice di 3 puntatori a funzioni che riceveranno come argomento un `int` e che restituiranno `void`". Il vettore è stato inizializzato con i nomi delle tre funzioni. Il valore, compreso tra 0 e 2, che l'utente immetterà sarà utilizzato come indice per il vettore di puntatori a funzioni. L'invocazione della funzione (riga 26) sarà quindi effettuata nel modo seguente:

```
(*f[ choice ]) ( choice );
```

In questa chiamata, `f[ choice ]` selezionerà il puntatore immagazzinato nella posizione `choice` del vettore. Una volta risolto il riferimento del puntatore, la funzione sarà invocata e `choice` gli sarà fornito come argomento. Ogni funzione visualizzerà il valore dell'argomento ricevuto e il proprio nome di funzione per indicare che sarà stata richiamata correttamente. Negli esercizi svilupperete un sistema guidato da menu.

```
1  /* Fig. 7.28: fig07_28.c
2     Dimostrazione di utilizzo di un vettore di puntatori a funzioni */
3  #include <stdio.h>
4
5  /* prototipi */
6  void function1( int a );
7  void function2( int b );
8  void function3( int c );
9
10 int main()
11 {
12    /* inizializza il vettore di 3 puntatori a funzioni
13       che prendono un argomento di tipo int e restituiscono void */
14    void (*f[ 3 ])( int ) = { function1, function2, function3 };
15
```

Figura 7.28 Dimostrazione di utilizzo di un vettore di puntatori a funzioni (continua)

```
16      int choice; /* variabile per memorizzare la scelta dell'utente */
17
18      printf( "Enter a number between 0 and 2, 3 to end: " );
19      scanf( "%d", &choice );
20
21      /* elabora la scelta dell'utente */
22      while ( choice >= 0 && choice < 3 ) {
23
24         /* invoca la funzione alla posizione indicata da choice
              nel vettore f
25            e passa choice come argomento */
26         (*f[ choice ])( choice );
27
28         printf( "Enter a number between 0 and 2, 3 to end: " );
29         scanf( "%d", &choice );
30      } /* fine del comando while */
31
32      printf( "Program execution completed.\n" );
33
34      return 0; /* indica che il programma è terminato con successo */
35
36   } /* fine della funzione main */
37
38   void function1( int a )
39   {
40      printf( "You entered %d so function1 was called\n\n", a );
41   } /* fine della funzione function1 */
42
43   void function2( int b )
44   {
45      printf( "You entered %d so function2 was called\n\n", b );
46   } /* fine della funzione function2 */
47
48    void function3( int c )
49   {
50      printf( "You entered %d so function3 was called\n\n", c );
51   } /* fine della funzione function3 */
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Figura 7.28 Dimostrazione di utilizzo di un vettore di puntatori a funzioni

Esercizi di autovalutazione

7.1 Rispondete a ognuna delle seguenti domande:

a) Un puntatore è una variabile che contiene come suo valore l'_____ di un'altra variabile.

b) I tre valori che potranno essere utilizzati per inizializzare un puntatore sono _____, _____ o _____.

c) L'unico intero che possa essere assegnato a un puntatore è _____.

7.2 Determinate se le seguenti affermazioni siano *vere* o *false*. Qualora la risposta sia *falsa*, spiegatene il motivo.

a) L'operatore di indirizzo `&` può essere applicato soltanto alle costanti, alle espressioni e alle variabili dichiarate con la specifica di classe di memoria `register`.

b) Il riferimento di un puntatore che sia stato dichiarato `void` può essere risolto.

c) I puntatori di tipo differente non possono essere assegnati gli uni agli altri senza un'operazione di conversione.

7.3 Rispondete a ognuna delle seguenti domande. Supponete che i numeri in virgola mobile con precisione singola siano immagazzinati in 4 byte e che l'indirizzo di partenza per il vettore corrisponda alla locazione di memoria 1002500. Ogni parte dell'esercizio, laddove appropriato, dovrà utilizzare i risultati di quelle precedenti.

a) Dichiarate il vettore di tipo `float` chiamato `numbers` contenente 10 elementi e inizializzateli con i valori `0,0`, `1,1`, `2,2`, ... `9,9`. Supponete che la costante simbolica `SIZE` sia stata definita con il testo di sostituzione `10`.

b) Dichiarate un puntatore `nPtr` che faccia riferimento a un oggetto di tipo `float`.

c) Visualizzate gli elementi del vettore `numbers` utilizzando la notazione con gli indici di vettore. Utilizzate un comando `for` e supponete che la variabile di controllo intera `i` sia già stata dichiarata. Visualizzate ogni numero con 1 posizione di precisione a destra della virgola decimale.

d) Fornite due istruzioni distinte che assegnino alla variabile di tipo puntatore `nPtr` l'indirizzo di partenza del vettore `numbers`.

e) Visualizzate gli elementi del vettore `numbers` utilizzando la notazione con puntatore e offset usando `nPtr` come puntatore.

f) Visualizzate gli elementi del vettore `numbers` utilizzando la notazione con puntatore e offset usando il nome del vettore come puntatore.

g) Visualizzate gli elementi del vettore `numbers` utilizzando gli indici applicati al puntatore `nPtr`.

h) Puntate all'elemento 4 del vettore `numbers`, utilizzando la notazione con gli indici di vettore, quella con puntatore e offset usando il nome del vettore come puntatore, quella con gli indici applicati al puntatore `nPtr`, e la notazione con puntatore e offset usando `nPtr`.

i) Supponendo che `nPtr` faccia riferimento all'inizio del vettore `numbers`, quale indirizzo sarebbe puntato da `nPtr + 8`? Quale valore sarebbe immagazzinato in quella locazione di memoria?

j) Supponendo che `nPtr` faccia riferimento a `numbers[ 5 ]`, quale indirizzo sarebbe puntato da `nPtr -= 4`? Quale sarebbe il valore immagazzinato in quella locazione di memoria?

7.4 Per ognuna delle seguenti attività scrivete una singola istruzione che esegua il compito indicato. Supponete che siano già state dichiarate le variabili in virgola mobile `number1` e `number2` e che `number1` sia stata inizializzata a `7,3`.

a) Dichiarate la variabile `fPtr` come puntatore a un oggetto di tipo `float`.

b) Assegnate l'indirizzo della variabile `number1` alla variabile di tipo puntatore `fPtr`.

c) Visualizzate il valore dell'oggetto puntato da `fPtr`.

d) Assegnate alla variabile `number2` il valore dell'oggetto puntato da `fPtr`.
e) Visualizzate il valore di `number2`.
f) Visualizzate l'indirizzo di `number1`. Utilizzate la specifica di conversione `%p`.
g) Visualizzate l'indirizzo immagazzinato in `fPtr`. Utilizzate la specifica di conversione `%p`. Il valore visualizzato è identico a quello dell'indirizzo di `number1`?

7.5 Eseguite ognuna delle seguenti attività.
a) Scrivete l'intestazione di una funzione chiamata `exchange` che riceva come parametri due puntatori ai numeri in virgola mobile `x` e `y` e non restituisca alcun valore.
b) Scrivete il prototipo per la funzione della parte (a).
c) Scrivete l'intestazione di una funzione chiamata `evaluate` che restituisca un intero e che riceva come argomenti l'intero `x` e un puntatore alla funzione `poly`. La funzione `poly` riceverà un parametro intero e restituirà un intero.
d) Scrivete il prototipo per la funzione della parte (c).

7.6 Trovate l'errore in ognuno dei seguenti segmenti di programma. Supponete che

```
int *zPtr;   /* zPtr punterà al vettore z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
sPtr = z;
```

a)
```
++zPtr;
```
b)
```
/* usa un puntatore per ottenere il primo valore del vettore */
number = zPtr;
```
c)
```
/* assegna a number l'elemento 2 del vettore (il valore 3) */
number = *zPtr[ 2 ];
```
d)
```
/* visualizza l'intero vettore z */
for ( i = 0; i <= 5; i++ )
   printf( "%d ", zPtr[ i ] );
```
e)
```
/* assegna a number il valore puntato da sPtr */
number = *sPtr;
```
f)
```
++z;
```

Risposte agli esercizi di autovalutazione

7.1 a) indirizzo. b) `0`, `NULL`, un indirizzo. c) `0`.

7.2 a) Falso. L'operatore di indirizzo può essere applicato soltanto alle variabili e non può essere applicato a quelle dichiarate con la classe di memoria `register`.
b) Falso. Il riferimento di un puntatore a `void` non può essere risolto, perché non c'è modo di sapere esattamente a quanti byte di memoria si debba puntare.
c) Falso. A un puntatore `void` potranno essere assegnati tutti gli altri tipi di puntatore e a questi potrà essere assegnato quello di tipo `void`.

7.3 a)
```
float numbers[ SIZE ] =
   { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
```
b)
```
float *nPtr;
```
c)
```
for ( i = 0; i < SIZE; i++ )
   printf( "%.1f ", numbers[ i ] );
```
d)
```
nPtr = numbers;
   nPtr = &numbers[ 0 ];
```

e)
```
for ( i = 0; i < SIZE; i++ )
   printf( "%.1f ", *( nPtr + i ) );
```
f)
```
for ( i = 0; i < SIZE; i++ )
   printf( "%.1f ", *( numbers + i ) );
```
g)
```
for ( i = 0; i < SIZE; i++ )
   printf( "%.1f ", nPtr[ i ] );
```
h)
```
numbers[ 4 ]
*( numbers + 4 )
 nPtr[ 4 ]
 *( nPtr + 4 )
```
i) L'indirizzo è `1002500 + 8 * 4 = 1002532`. Il valore è `8,8`.

j) L'indirizzo di `numbers[ 5 ]` è `1002500 + 5 * 4 = 1002520`.
L'indirizzo di `nPtr -= 4` è `1002520 - 4 * 4 = 1002504`.
Il valore in quella locazione è `1,1`.

7.4 a)
```
float *fPtr;
```
b)
```
fPtr = &number1;
```
c)
```
printf( "The value of *fPtr is %f\n", *fPtr );
```
d)
```
number2 = *fPtr;
```
e)
```
printf( "The value of number2 is %f\n", number2 );
```
f)
```
printf( "The address of number1 is %p\n", &number1 );
```
g)
```
printf( "The address stored in fPtr is %p\n", fPtr );
```
Sì, il valore è lo stesso.

7.5 a)
```
void exchange( float *x, float *y )
```
b)
```
void exchange( float *x, float *y );
```
c)
```
int evaluate( int x, int (*poly)( int ) )
```
d)
```
int evaluate( int x, int (*poly)( int ) );
```

7.6 a) Errore: `zPtr` non è stato inizializzato.
Correzione: inizializzate `zPtr` con `zPtr = z;`

b) Errore: il riferimento del puntatore non è stato risolto.
Correzione: cambiate l'istruzione in `number = *zPtr;`

c) Errore: `zPtr[ 2 ]` non è un puntatore e quindi non c'è nessun riferimento da risolvere.
Correzione: cambiate `*zPtr[ 2 ]` in `zPtr[ 2 ]`.

d) Errore: con l'indice del puntatore si fa riferimento a un elemento esterno ai limiti del vettore.
Correzione: cambiate l'operatore <= nella condizione del `for` in <.

e) Errore: tentativo di risolvere il riferimento di un puntatore a `void`.
Correzione: per risolvere il riferimento del puntatore questo dovrà prima essere convertito in un puntatore a interi. Cambiate la suddetta istruzione in `number = *( int * ) sPtr;`

f) Errore: tentativo di modificare il nome di un vettore con l'aritmetica dei puntatori.
Correzione: utilizzate un puntatore invece del nome del vettore per eseguire l'operazione di aritmetica dei puntatori, oppure utilizzate gli indici con il nome del vettore per fare riferimento a un elemento specifico.

Esercizi

7.7 Rispondete a ognuna delle seguenti domande:

a) L'operatore _____ restituisce la locazione di memoria in cui è immagazzinato il suo operando.

b) L'operatore _____ restituisce il valore dell'oggetto puntato dal proprio operando.

c) Per simulare una chiamata per riferimento, qualora si passi a una funzione una variabile che non sia un vettore, sarà necessario passare alla funzione l'_____ della variabile.

7.8 Determinate se le seguenti affermazioni siano *vere* o *false*. Qualora siano *false* spiegatene il motivo.

a) Non ha senso confrontare due puntatori che facciano riferimento a vettori distinti.
b) Poiché il nome di un vettore è un puntatore al suo primo elemento, i nomi dei vettori possono essere manipolati nello stesso modo dei puntatori.

7.9 Rispondete a ognuna delle seguenti domande. Supponete che gli interi senza segno siano immagazzinati in 2 byte e che l'indirizzo di partenza per il vettore sia alla locazione di memoria 1002500.

a) Dichiarate il vettore di tipo `unsigned int` chiamato `values` contenente cinque elementi e inizializzateli con gli interi pari compresi tra 2 a 10. Supponete che la costante simbolica `SIZE` sia già stata definita con il testo di sostituzione `5`.
b) Dichiarate un puntatore `vPtr` che faccia riferimento a un oggetto di tipo `unsigned int`.
c) Visualizzate gli elementi di `values` utilizzando la notazione con gli indici di vettore. Usate un comando `for` e supponete che la variabile di controllo intera `i` sia già stata dichiarata.
d) Fornite due distinte istruzioni che assegnino alla variabile di tipo puntatore `vPtr` l'indirizzo di partenza del vettore `values`.
e) Visualizzate gli elementi del vettore `values` utilizzando la notazione con puntatore e offset.
f) Visualizzate gli elementi del vettore `values` utilizzando la notazione con puntatore e offset e usando come puntatore il nome del vettore.
g) Visualizzate gli elementi del vettore `values` utilizzando gli indici con un puntatore che faccia riferimento al vettore stesso.
h) Puntate all'elemento 5 del vettore `values` utilizzando la notazione con gli indici di vettore, quella con puntatore e offset usando il nome del vettore come puntatore, quella con gli indici di puntatore e quella con puntatore e offset.
i) Quale indirizzo sarebbe puntato da `vPtr + 3`? Quale valore sarebbe immagazzinato in quella locazione?
j) Supponendo che `vPtr` faccia riferimento a `values[ 4 ]` quale indirizzo sarebbe puntato da `vPtr -= 4`? Quale valore sarebbe immagazzinato in quella locazione?

7.10 Per ognuna delle seguenti attività scrivete una singola istruzione che esegua il compito indicato. Supponete che siano già state dichiarate le variabili intere `long value1` e `value2` e che `value1` sia stata inizializzata a `200000`.

a) Dichiarate la variabile `lPtr` come puntatore a un oggetto di tipo `long`.
b) Assegnate l'indirizzo di `value1` alla variabile di tipo puntatore `lPtr`.
c) Visualizzate il valore dell'oggetto puntato da `lPtr`.
d) Assegnate il valore dell'oggetto puntato da `lPtr` alla variabile `value2`.
e) Visualizzate il valore di `value2`.
f) Visualizzate l'indirizzo di `value1`.
g) Visualizzate l'indirizzo immagazzinato in `lPtr`. L'indirizzo visualizzato è identico a quello di `value1`?

7.11 Eseguite ognuna delle seguenti attività.

a) Scrivete l'intestazione della funzione `zero` che avrà come parametro un vettore `bigIntegers` di interi `long` e non restituirà alcun valore.
b) Scrivete il prototipo per la funzione della *Parte a.*
c) Scrivete l'intestazione della funzione `add1AndSum` che avrà come parametro un vettore di interi `oneTooSmall` e restituirà un intero.
d) Scrivete il prototipo per la funzione descritta nella *Parte c.*

Nota: gli Esercizi dal 7.12 al 7.15 sono abbastanza impegnativi. Una volta che avrete risolto questi problemi, dovreste essere in grado di implementare facilmente i giochi di carte più comuni.

7.12 Modificate il programma della Figura 7.24, in modo che la funzione di distribuzione dispensi le cinque carte di una mano di poker. In seguito scrivete le seguenti funzioni aggiuntive:

a) Determinate se la mano contenga una coppia.
b) Determinate se la mano contenga una doppia coppia.
c) Determinate se la mano contenga un tris (per esempio, tre fanti).
d) Determinate se la mano contenga un poker (per esempio, quattro assi).
e) Determinate se la mano contenga un colore (ovverosia, cinque carte dello stesso seme).
f) Determinate se la mano contenga una scala (ovverosia, cinque carte con valori consecutivi).

7.13 Utilizzate le funzioni sviluppate nell'Esercizio 7.12 per scrivere un programma che distribuisca le cinque carte di due mani di poker, le valuti e determini qual è la mano migliore.

7.14 Modificate il programma della Figura 7.13 in modo da simulare un mazziere. Le cinque carte della mano del mazziere saranno distribuite "a faccia in giù" così che il giocatore non le possa vedere. Il programma dovrà quindi valutare la mano del mazziere e, basandosi sulla qualità di quella, dovrà estrarre altre carte per sostituire quelle scartate dalla mano originaria. Il programma dovrà quindi rivalutare la mano del mazziere. [*Attenzione*: questo è un problema difficile!]

7.15 Modificate il programma sviluppato nell'Esercizio 7.14 così che possa gestire automaticamente la mano del mazziere, mentre consenta al giocatore di decidere quali carte della sua mano sostituire. Il programma dovrà quindi valutare entrambe le mani e determinare chi avrà vinto. Utilizzate ora questo nuovo programma per giocare 20 partite contro il computer. Chi ne vince di più, voi o il computer? Fate giocare uno dei vostri amici in 20 partite contro il computer. Chi ne vince più? Basandovi sui risultati di queste partite, apportate le opportune modifiche per raffinare il vostro programma di gioco (anche questo è un problema difficile). Giocate altre 20 partite. Gioca meglio il vostro nuovo programma?

7.16 Nel programma per il mescolamento e la distribuzione delle carte della Figura 7.24, abbiamo intenzionalmente utilizzato un algoritmo di mescolamento inefficiente che ha introdotto la possibilità di differimenti indefiniti. In questo esercizio creerete un algoritmo di mescolamento ad alta efficienza che eviterà il differimento indefinito.

Modificate il programma della Figura 7.24 nel modo seguente. Inizializzate la matrice `deck` come mostrato nella Figura 7.29. Modificate la funzione shuffle in modo che iteri su ogni riga e colonna della matrice, toccando ognuno degli elementi una sola volta. Ogni elemento dovrà essere scambiato con un altro selezionato a caso dalla matrice.

Visualizzate la matrice risultante così che possiate determinare se il mazzo di carte sia stato mescolato in modo soddisfacente (come nella Figura 7.30, per esempio). Per assicurarvi un mescolamento soddisfacente potreste anche fare in modo che il programma richiami più volte la funzione shuffle.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 7.29 La matrice deck non mescolata

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 7.30 Esempio di matrice deck mescolata

Osservate che sebbene l'approccio utilizzato in questo problema migliori l'algoritmo di mescolamento, quello di distribuzione richiederà ancora delle ricerche nella matrice `deck`per la carta 1, poi per la 2, poi per la 3 e così via. Come se non bastasse, l'algoritmo di distribuzione continuerà a cercare una carta nel mazzo anche dopo averla già ritrovata e distribuita. Modificate il programma della Figura 7.24 in modo che, una volta che abbia distribuito una carta, non faccia ulteriori tentativi per ritrovarla e che proceda immediatamente con la distribuzione di quella successiva. Nel Capitolo 10 svilupperemo un algoritmo che richiederà una sola operazione per carta.

7.17 (*Simulazione: la tartaruga e la lepre*) In questo esercizio ricreerete uno dei momenti "storici" più importanti, vale a dire la classica corsa tra la tartaruga e la lepre. Utilizzerete la generazione dei numeri casuali per sviluppare una simulazione di questo memorabile evento.

I nostri contendenti cominceranno la loro corsa nella "casella 1" delle 70 che compongono il percorso della gara. Ogni casella rappresenta una delle posizioni possibili lungo il percorso della gara. La linea di arrivo è nella casella numero 70. Il primo contendente che avrà raggiunto o superato la casella 70 sarà premiato con un secchio di carote e lattughe fresche. Il percorso si inerpica serpeggiando sul fianco di una montagna sdrucciolevole, perciò i contendenti potranno perdere terreno di quando in quando.

C'è un orologio che scandisce ogni secondo. A ogni tic dell'orologio il vostro programma dovrà aggiustare la posizione degli animali secondo le regole in Figura 7.31.

Animale	Tipo di mossa	Percentuale di tempo	Mossa effettiva
Tartaruga	Arrancata rapida	50%	3 caselle a destra
	Scivolone	20%	6 caselle a sinistra
	Arrancata lenta	30%	1 casella a destra
Lepre	Dormita	20%	Nessuna mossa
	Salto lungo	20%	9 caselle a destra
	Scivolone lungo	10%	12 caselle a sinistra
	Salto corto	30%	1 casella a destra
	Scivolone corto	20%	2 caselle a sinistra

Figura 7.31 Regole della tartaruga e della lepre per aggiustare la posizione

Utilizzate le variabili per mantenere traccia delle posizioni degli animali (ovverosia dei numeri da 1 a 70). Fate partire ogni animale dalla posizione 1 (vale a dire dalle "gabbie di partenza"). Riportate nella casella 1 l'animale che dovesse eventualmente scivolare indietro in una posizione precedente a quella.

Generate le percentuali mostrate nella tabella precedente producendo un intero casuale i compreso nell'intervallo $1 \leq i \leq 10$. Nel caso della tartaruga eseguirete una "arrancata rapida" quando $1 \leq i \leq 5$, una "scivolata" quando $6 \leq i \leq 7$ o una "arrancata lenta" quando $8 \leq i \leq 10$. Utilizzate una tecnica simile per far correre la lepre.

Cominciate la gara visualizzando:

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

In seguito, per ogni tic dell'orologio (ovverosia per ciascuna ripetizione del ciclo), visualizzerete una linea di 70 posizioni che mostri una lettera `T` in quella della tartaruga e una `L` in quella della lepre. Di quando in quando, i contendenti si ritroveranno nella stessa casella. In tal caso la tartaruga morderà la lepre e il vostro programma dovrà visualizzare `OUCH!!!` in quella posizione. Tutte le posizioni di visualizzazione diverse da `T`, `L` od `OUCH!!!` dovranno restare vuote.

Dopo che ogni linea sarà stata visualizzata, controllerete se uno dei due animali abbia raggiunto o superato la casella 70. In caso affermativo visualizzerete il nome del vincitore e terminerete la simulazione. Nel caso che abbia vinto la tartaruga visualizzerete "`TORTOISE WINS!!! YAY!!!`". Nel caso che abbia vinto la lepre visualizzerete "`Hare wins. Yuch.`" Nel caso che entrambi gli animali tagliassero il traguardo nello stesso tic dell'orologio, potreste favorire la tartaruga (poverina), oppure potreste visualizzare "`It's a tie.`" Nel caso che nessuno dei due animali abbia ancora vinto, eseguite un'altra volta il ciclo per simulare il successivo tic dell'orologio. Radunate un gruppo di tifosi che assistano alla gara, quando sarete pronti a mandare in esecuzione il vostro programma. Il coinvolgimento del vostro pubblico vi sorprenderà.

Sezione speciale: costruite il vostro computer

Nei prossimi esercizi abbandoneremo temporaneamente il mondo della programmazione con linguaggi di alto livello. "Sbucceremo" un computer e daremo uno sguardo alla sua struttura interna. Introdurremo la programmazione in linguaggio macchina e con questo scriveremo diversi programmi. In seguito, per fare in modo che sia un'esperienza di particolare valore, costruiremo un computer (attraverso la tecnica della *simulazione* software) sul quale potrete far eseguire i vostri programmi scritti in linguaggio macchina!

7.18 (*Programmazione in linguaggio macchina*) Creeremo ora un computer che chiameremo Simpletron. Come il suo nome lascia intendere si tratta di una macchina semplice, ma come vedremo presto anche potente. Il Simpletron eseguirà dei programmi scritti nell'unico linguaggio che sia in grado di comprendere direttamente, vale a dire il Linguaggio Macchina del Simpletron, che abbrevieremo in LMS.

Il Simpletron contiene un *accumulatore*: un "registro speciale" in cui saranno inserite tutte le informazioni, prima che il Simpletron possa utilizzarle per i calcoli o per esaminarle in vari modi. Tutte le informazioni saranno manipolate all'interno del Simpletron come *parole*. Una parola è un numero decimale di quattro cifre con segno, come `+3364`, `-1293`, `+0007`, `-0001` ecc. Il Simpletron è equipaggiato con una memoria di 100 parole alle quali faremo riferimento attraverso i loro numeri di locazione `00`, `01`, ... `99`.

Prima di poter eseguire un programma LMS dovremo *caricarlo* o immetterlo nella memoria. La prima istruzione (o comando) di ogni programma LMS sarà sempre caricata nella posizione `00`.

Ogni istruzione scritta in LMS occuperà una parola nella memoria del Simpletron e, di conseguenza, corrisponderà a un numero decimale di quattro cifre con segno. Ovviamente il segno di ogni istruzione LMS sarà sempre quello positivo, mentre quelli delle parole che contengano dei dati potranno essere positivi o negativi. Ogni locazione della memoria del Simpletron potrà contenere un'istruzione, il valore di un dato utilizzato dal programma, oppure un'area di memoria inutilizzata (e quindi indefinita). Le prime due cifre di ogni istruzione LMS corrisponderanno al *codice dell'operazione* che dovrà essere eseguita. I codici delle operazioni LMS sono riassunti nella Figura 7.32.

Codice dell'operazione	Significato
Operazioni di input/output:	
`#define READ 10`	Legge una parola dal terminale e la immagazzina in una specifica locazione di memoria.
`#define WRITE 11`	Scrive sul terminale la parola contenuta in una specifica locazione di memoria.
Operazioni di caricamento/immagazzinamento:	
`#define LOAD 20`	Carica nell'accumulatore la parola contenuta in una specifica locazione di memoria.
`#define STORE 21`	Archivia il contenuto dell'accumulatore in una specifica locazione di memoria.
Operazioni aritmetiche:	
`#define ADD 30`	Aggiunge la parola contenuta in una specifica locazione di memoria a quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define SUBTRACT 31`	Sottrae la parola contenuta in una specifica locazione di memoria da quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define DIVIDE 32`	Divide la parola contenuta in una specifica locazione di memoria per quella contenuta nell'accumulatore (lasciando in questo il risultato).
`#define MULTIPLY 33`	Moltiplica la parola contenuta in una specifica locazione di memoria per quella contenuta nell'accumulatore (lasciando in questo il risultato).
Operazioni di trasferimento del controllo:	
`#define BRANCH 40`	Salta a una specifica locazione di memoria.
`#define BRANCHNEG 41`	Salta a una specifica locazione di memoria, se l'accumulatore contiene un valore negativo.
`#define BRANCHZERO 42`	Salta a una specifica locazione di memoria, se l'accumulatore contiene un valore uguale a zero.
`#define HALT 43`	Ferma l'esecuzione del programma.

Figura 7.32 I codici di operazione del Linguaggio Macchina del Simpletron (LMS)

Le ultime due cifre di un'istruzione LMS rappresenteranno invece l'*operando*, ovverosia l'indirizzo della locazione di memoria che conterrà la parola su cui l'operazione sarà applicata. Consideriamo ora alcuni semplici programmi in LMS.

Esempio 1 Locazione	Numero	Istruzione
00	+1007	(Legge A)
01	+1008	(Legge B)
02	+2007	(Carica A nell'accumulatore)
03	+3008	(Somma B all'accumulatore)
04	+2109	(Memorizza il valore dell'accumulatore in C)
05	+1109	(Stampa C)
06	+4300	(Halt)
07	+0000	(Variabile A)
08	+0000	(Variabile B)
09	+0000	(Risultato C)

Questo programma in LMS leggerà due numeri dalla tastiera e calcolerà e visualizzerà la loro somma. L'istruzione `+1007` leggerà il primo numero dalla tastiera e lo inserirà nella locazione di memoria `07` (che sarà già stata azzerata). In seguito `+1008` leggerà il secondo numero nella locazione `08`. L'istruzione *load*, `+2007`, sistemerà il primo numero nell'accumulatore, mentre l'istruzione *add*, `+3008`, aggiungerà il secondo numero a quello contenuto nell'accumulatore. *Tutte le operazioni aritmetiche del linguaggio LMS lasciano il loro risultato nell'accumulatore.* L'istruzione *store*, `+2109`, riporterà il risultato nella locazione di memoria `09` dalla quale l'istruzione *write*, `+1109`, lo preleverà e lo visualizzerà come un numero decimale di quattro cifre con segno. L'istruzione *halt*, `+4300`, terminerà l'esecuzione del programma.

Esempio 2 Locazione	Numero	Istruzione
00	+1009	(Legge A)
01	+1010	(Legge B)
02	+2009	(Carica A nell'accumulatore)
03	+3110	(Sottrae B dall'accumulatore)
04	+4107	(Salta a 07 se l'accumulatore è negativo)
05	+1109	(Stampa A)
06	+4300	(Halt)
07	+1110	(Stampa B)
08	+4300	(Halt)
09	+0000	(Variabile A)
10	+0000	(Variabile B)

Il precedente programma LMS leggerà due numeri dalla tastiera e determinerà e visualizzerà quello maggiore. Osservate l'uso della istruzione `+4107` come un trasferimento di controllo condizionale molto simile all'istruzione `if` del C. Scrivete ora dei programmi in LMS che eseguano ognuna delle seguenti attività.

a) Utilizzate un ciclo controllato da un valore sentinella per leggere 10 numeri positivi e calcolare e visualizzare la loro somma.
b) Utilizzate un ciclo controllato da un contatore per leggere sette numeri, positivi e negativi, e calcolate e visualizzate la loro media.
c) Leggete una serie di numeri e determinate e visualizzate quello maggiore. Il primo numero letto indicherà quanti valori dovranno essere elaborati.

7.19 (*Un simulatore di computer*) Potrà sembrare esagerato, ma con questo esercizio costruirete il vostro computer. No, non dovrete saldare dei componenti. Utilizzerete piuttosto la potente tecnica della *simulazione software* per creare un *modello software* del Simpletron. Non rimarrete delusi. Il vostro simulatore Simpletron trasformerà il computer che state utilizzando in un Simpletron e sarete effettivamente in grado di eseguire, provare e mettere a punto i programmi LMS che avete scritto nell'esercizio 7.18.

Nel momento in cui eseguirete il vostro simulatore Simpletron questo dovrà incominciare visualizzando:

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Simulate la memoria del Simpletron con il vettore unidimensionale `memory` di 100 elementi. Supponete che il simulatore sia già in esecuzione ed esaminiamo il dialogo che si svilupperà con esso, man mano che immettiamo il programma mostrato nell'Esempio 2 dell'Esercizio 7.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins  ***
```

Ora che il programma LMS è stato immesso (o caricato) nel vettore `memory`, il Simpletron provvederà a eseguirlo. L'esecuzione comincerà con l'istruzione nella locazione `00` e, come in C, continuerà in modo sequenziale, sempre che non si dirami in qualche altra parte del programma a causa di un trasferimento di controllo.

Utilizzate la variabile `accumulator` per rappresentare il registro accumulatore. Usate la variabile `instructionCounter` per conservare l'indirizzo di memoria in cui sarà contenuta l'istruzione da eseguire. Utilizzate la variabile `operationCode` per indicare l'operazione da eseguire, ovverosia le due cifre a sinistra nella parola dell'istruzione. Usate la variabile `operand` per indicare la locazione di memoria su cui opererà l'istruzione corrente. In altri termini, `operand` corrisponderà alle due cifre più a destra dell'istruzione da eseguire. Non eseguite direttamente le istruzioni contenute nella memoria. Trasferite piuttosto quella da eseguire dalla memoria in una variabile chiamata `instructionRegister`. In seguito "staccherete" le due cifre di sinistra per sistemarle in `operationCode` e "separerete" le due cifre di destra per sistemarle in `operand`.

Nel momento in cui il Simpletron comincerà l'esecuzione i registri speciali saranno dunque inizializzati nel modo seguente:

```
accumulator           +0000
instructionCounter      00
instructionRegister   +0000
operationCode           00
operand                 00
```

Ora "seguiamo" l'esecuzione della prima istruzione LMS: il `+1009` sistemato nella locazione di memoria `00`. Quello che seguiremo è detto *ciclo di esecuzione dell'istruzione.*

La variabile `instructionCounter` ci indica la locazione della prossima istruzione da eseguire. "Preleveremo" dunque il contenuto di quella posizione dal vettore `memory` usando l'istruzione C:

```
instructionRegister = memory[ instructionCounter ];
```

Il codice dell'operazione e l'operando saranno estratti dal registro delle istruzioni con:

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Ora il Simpletron è in grado di determinare che il codice dell'operazione corrisponde in realtà a una *read* (e non a una *write*, a una *load* ecc.). Un comando `switch` distinguerà le dodici operazioni del linguaggio LMS.

All'interno del comando `switch` il comportamento delle varie istruzioni sarà simulato nel modo seguente (lasciamo a voi le altre istruzioni):

```
read:  scanf("%d", &memory[ operand ]);
load:  accumulator = memory[ operand ];
add:   accumulator += memory[ operand ];
Varie istruzioni di salto: ne discuteremo tra breve.
halt:  Questa istruzione visualizza il messaggio
       *** Simpletron execution terminated ***
```

e in seguito visualizzerà il nome e il contenuto di tutti i registri, così come quello di tutta la memoria. Un output di questo genere è spesso chiamato *dump del computer.* Per aiutarvi a implementare la vostra funzione di dump nella Figura 7.33 abbiamo riportato un esempio per il formato del dump. Osservate che un dump eseguito alla fine dell'esecuzione di un programma Simpletron dovrebbe mostrare i valori correnti delle istruzioni e dei dati, così come sono in quel momento.

Procediamo con l'esecuzione della prima istruzione del nostro programma, vale a dire la `+1009` della posizione `00`. Come abbiamo affermato prima, il comando `switch` la simulerà eseguendo l'istruzione C

```
scanf( "%d", &memory[ operand ] );
```

Prima che la funzione `scanf` sia eseguita, dovrà essere visualizzato un punto interrogativo (?) per richiedere l'input dell'utente. Il Simpletron attenderà che l'utente immetta un valore e prema il *tasto invio.* A quel punto il valore sarà sistemato nella locazione `09`.

La simulazione della prima istruzione è stata finalmente completata. Ci rimane solo la preparazione del Simpletron all'esecuzione della prossima istruzione. Dato che l'istruzione appena eseguita non era un trasferimento di controllo, avremo semplicemente bisogno di incrementare il registro contatore delle istruzioni, nel modo seguente:

```
++instructionCounter;
```

Questo passo completa davvero l'esecuzione simulata della prima istruzione. L'intero processo (ovverosia il ciclo di esecuzione dell'istruzione) ricomincerà nuovamente con il recupero della prossima istruzione da eseguire.

```
REGISTERS:
accumulator +0000
instructionCounter                    00
instructionRegister     +0000
operationCode     00
operand     00

MEMORY:
      0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Figura 7.33 Un esempio di dump della memoria del Simpletron

Ora esaminiamo in che modo saranno simulate le istruzioni di salto, ovverosia i trasferimenti di controllo. In effetti, sarà necessario solo aggiornare in modo appropriato il valore contenuto nel contatore di istruzioni. Di conseguenza, l'istruzione di salto incondizionato (`40`) sarà simulata all'interno del comando `switch` con:

```
instructionCounter = operand;
```

L'istruzione condizionale "salto se l'accumulatore è uguale a zero" sarà simulata con:

```
if ( accumulator == 0 )
   instructionCounter = operand;
```

A questo punto, dovreste essere in grado di implementare il vostro simulatore di Simpletron e di eseguire ognuno dei programmi LMS che avete scritto nell'Esercizio 7.18. Potrete abbellire il linguaggio LMS con delle caratteristiche aggiuntive e implementarle nel vostro simulatore.

Il vostro simulatore dovrà effettuare dei controlli per vari tipi di errore. Per esempio, durante la fase di caricamento del programma, ogni numero che l'utente tenterà di immettere nel vettore `memory` del Simpletron dovrà essere compreso nell'intervallo da `-9999` a `+9999`. Il vostro simulatore dovrà quindi utilizzare un ciclo `while` per controllare che ogni numero immesso sia compreso in quell'intervallo e, in caso contrario, continuare a richiederlo all'utente finché non ne avrà immesso uno corretto.

Durante la fase di esecuzione, il vostro simulatore dovrà effettuare dei controlli per vari errori gravi, come i tentativi di eseguire delle divisioni per zero o di eseguire dei codici di operazione non validi, il superamento della capacità dell'accumulatore (ovverosia le operazioni aritmetiche che producono valori maggiori di `+9999` o minori di `-9999`). Insomma, occorrerà gestire gli *errori fatali*. Nel momento in cui avrà intercettato un errore fatale, il vostro simulatore dovrà visualizzare un messaggio di errore come:

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

e visualizzare un dump completo del computer nel formato di cui abbiamo discusso in precedenza. Ciò aiuterà l'utente a individuare l'errore del programma.

7.20 Modificate il programma di mescolamento e distribuzione delle carte proposto nella Figura 7.24, in modo che le suddette operazioni siano eseguite da una funzione unica (`shuffleAndDeal`). Questa dovrà contenere una struttura di iterazione nidificata simile a quella della funzione `shuffle` mostrata nella Figura 7.24.

7.21 Che cosa farà questo programma?

```
/* ex07_21.c */
/* Che cosa farà questo programma? */
#include <stdio.h>

void mystery1( char *s1, const char *s2 ); /* prototipo */

int main()
{
   char string1[ 80 ]; /* crea un vettore di caratteri */
   char string2[ 80 ]; /* crea un vettore di caratteri */

   printf( "Enter two strings: " );
   scanf( "%s%s", string1, string2 );

   mystery1( string1, string2 );

   printf( "%s\n", string1 );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */

/* Che cosa farà questa funzione? */
void mystery1( char *s1, const char *s2 )
{
   while ( *s1 != '\0' ) {
      ++s1;
   } /* fine del comando while */

   for (  ; *s1 = *s2; s1++, s2++ ) {
      ;   /* istruzione vuota */
   } /* fine del comando for */

} /* fine della funzione mystery1 */
```

7.22 Che cosa farà questo programma?

```
/* ex07_22.c */
/* Che cosa farà questo programma? */
#include <stdio.h>

int mystery2( const char *s ); /* prototipo */

int main()
{
   char string[ 80 ]; /* crea un vettore di caratteri */

   printf( "Enter a string: " );
   scanf( "%s", string );

   printf( "%d\n", mystery2( string ) );
```

```
15
16      return 0; /* indica che il programma è terminato con successo */
17   } /* fine della funzione main */
18
19   /* Che cosa farà questa funzione? */
20   int mystery2( const char *s )
21   {
22      int x; /* contatore */
23
24      /* itera scorrendo la stringa */25      for ( x = 0; *s != '\0'; s++ )
{
26         x++;
27      } /* fine del comando for */
28
29      return x;
30
31   } /* fine della funzione mystery2 */
```

7.23 Trovate l'errore in ognuno dei seguenti segmenti di programma. Spiegate in che modo sarà possibile correggerlo, sempre che lo sia.

a)
```
int *number;
printf( "%d\n", *number );
```

b)
```
float *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
int count;
for (  ; *s != '\0'; s++ )
   printf( "%c ", *s );
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
float x = 19.34;
float xPtr = &x;
printf( "%f\n", xPtr );
```

g)
```
char *s;
printf( "%s\n", s );
```

7.24 (*Quicksort*) Negli esempi e negli esercizi del Capitolo 6 abbiamo discusso le tecniche di ordinamento a bolle (bubble sort), di bucket sort e per selezione (selection sort). Presenteremo ora una tecnica ricorsiva di ordinamento chiamata Quicksort (ordinamento veloce). L'algoritmo fondamentale per un vettore di valori unidimensionale è il seguente:

a) *Passo di ripartizione:* prendete il primo elemento di un vettore disordinato e determinate la sua posizione finale in quello ordinato (cioè, quando tutti i valori del sottovettore sinistro, rispetto all'elemento, gli saranno inferiori e tutti quelli del sottovettore destro gli saranno superiori). A questo punto avremo un elemento sistemato nella sua giusta posizione e due sottovettori disordinati.

b) *Passo ricorsivo:* eseguite il passo 1 su ogni sottovettore disordinato.

Ogni volta che il passo 1 sarà eseguito su un sottovettore, un altro elemento sarà sistemato nella sua posizione finale all'interno del vettore ordinato e saranno creati due sottovettori disordinati. Considereremo sicuramente ordinato un sottovettore che sia formato da un solo elemento e, di conseguenza, questo sarà già nella sua posizione finale.

L'algoritmo di base sembra abbastanza semplice, ma in che modo determineremo la posizione finale del primo elemento di ogni sottovettore? Come esempio, consideriamo il seguente gruppo di valori (l'elemento in grassetto è quello utilizzato per la ripartizione, ovverosia, quello che sarà sistemato nella sua posizione finale all'interno del vettore ordinato):

37 2 6 4 89 8 10 12 68 45

a) Partendo da quello più a destra nel vettore, confrontate ogni elemento con il 37 finché non ne troviate uno minore che scambierete di posto con il 37. Il primo elemento minore di 37 è 12, perciò 37 e 12 saranno scambiati di posto. Il nuovo vettore sarà:

   12 2 6 4 89 8 10 37 68 45

   L'elemento 12 è in corsivo per ricordare che è stato appena scambiato di posto con il 37.

b) Partendo dalla sinistra del vettore, ma cominciando da quello successivo al 12, confrontate ogni elemento con il 37 finché non ne troviate uno maggiore che scambierete di posto con il 37. Il primo elemento maggiore di 37 è 89, perciò 37 e 89 saranno scambiati di posto. Il nuovo vettore sarà:

   12 2 6 4 37 8 10 *89* 68 45

c) Partendo dalla destra, ma cominciando da quello precedente a 89, confrontate ogni elemento con il 37 finché non ne troviate uno minore che scambierete di posto con il 37. Il primo elemento minore di 37 è 10, perciò 37 e 10 saranno scambiati di posto. Il nuovo vettore sarà:

   12 2 6 4 *10* 8 37 89 68 45

d) Partendo dalla sinistra, ma cominciando da quello successivo al 10, confrontate ogni elemento con il 37 finché non ne troviate uno maggiore che scambierete di posto con il 37. In questo caso non ci sono altri elementi maggiori di 37 perciò, confrontando il 37 con se stesso, sapremo che sarà già stato sistemato nella sua posizione finale all'interno del vettore ordinato.

Una volta che al suddetto vettore sarà stata applicata la ripartizione, ci saranno due sottovettori disordinati. Il sottovettore dei valori minori di 37 conterrà 12, 2, 6, 4, 10 e 8, mentre quello dei valori maggiori di 37 conterrà 89, 68 e 45. L'algoritmo di ordinamento dovrà dunque procedere con la ripartizione di entrambi i sottovettori nello stesso modo utilizzato per ripartire quello originale.

Scrivete una funzione ricorsiva `quicksort` che ordini un vettore unidimensionale di valori interi. La funzione dovrà ricevere come argomenti un vettore di valori interi, un indice di partenza e uno di fine. La funzione `partition` dovrà essere richiamata da `quicksort` per eseguire il passo di ripartizione.

7.25 (*Attraversamento di un labirinto*) La griglia seguente è la rappresentazione di un labirinto all'interno di una matrice.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

I simboli # e i punti (.) rappresentano rispettivamente le pareti e i corridoi del labirinto.

Esiste un semplice algoritmo di attraversamento di un labirinto che garantisce il ritrovamento dell'uscita (sempre che ce ne sia una). Nel caso che l'uscita non ci fosse vi ritrovereste di nuovo al punto di partenza. Poggiate la mano destra sulla parete alla vostra destra e cominciate a camminare in avanti. Non rimuovete mai la vostra mano dalla parete. Se il labirinto svolta a destra, seguite il muro a destra. Alla fine arriverete sicuramente all'uscita del labirinto, se nel frattempo non avrete rimosso la vostra mano dalla parete. È probabile che esista un percorso più breve di quello che avete intrapreso, ma in questo modo avrete la certezza di uscire dal labirinto.

Scrivete la funzione ricorsiva `mazeTraverse` per attraversare il labirinto. La funzione dovrà ricevere come argomenti una matrice di caratteri 12 per 12, per rappresentare il labirinto, e la posizione di partenza all'interno dello stesso. Man mano che `mazeTraverse` tenterà di individuare l'uscita dal labirinto, dovrà inserire il carattere `x` in ogni casella del percorso. La funzione dovrà visualizzare il labirinto dopo ogni mossa così che l'utente possa seguire la risoluzione del labirinto.

7.26 (*Generazione casuale di labirinti*) Scrivete una funzione `mazeGenerator` che riceva come argomento una matrice di caratteri 12 per 12 e produca un labirinto a caso. La funzione dovrà anche restituire i punti di ingresso e di uscita del labirinto. Provate la vostra funzione `mazeTraverse` dell'Esercizio 7.25 utilizzando vari labirinti generati a caso.

7.27 (*Labirinti di qualsiasi dimensione*) Generalizzate le funzioni `mazeTraverse` e `mazeGenerator` degli Esercizi 7.25 e 7.26, in modo da elaborare labirinti di qualsiasi larghezza e altezza.

7.28 (*Vettori di puntatori a funzioni*) Riscrivete il programma della Figura 6.22 in modo da utilizzare un'interfaccia guidata da menu. Il programma dovrà offrire all'utente le seguenti 4 opzioni:

```
Enter a choice:
  0  Print the array of grades
  1  Find the minimum grade
  2  Find the maximum grade
  3  Print the average on all tests for each student
  4  End program
```

Una restrizione all'utilizzo di vettori di puntatori a funzioni è che tutti i puntatori devono essere dello stesso tipo. Di conseguenza, devono essere dei puntatori a funzioni che restituiscano tutte lo stesso tipo di dato e che ricevano argomenti dello stesso tipo. Per questa ragione, le funzioni nella Figura 6.22 dovranno essere modificate in modo che restituiscano lo stesso tipo di dato e ricevano i medesimi parametri. Modificate le funzioni `minimum` e `maximum` in modo che visualizzino il valore minimo e quello massimo e non restituiscano niente. Per l'opzione 3, modificate la funzione `average` della Figura 6.22 in modo che invii in output la media di ogni studente (invece che quella di uno specifico). La funzione `average` non dovrà restituire nessun dato e riceverà gli stessi parametri di `printArray`, `minimum` e `maximum`. Immagazzinate i puntatori alle quattro funzioni nel vettore `processGrades` e utilizzate la scelta effettuata dall'utente come indice di vettore per richiamare le funzioni.

7.29 (*Modifiche al simulatore Simpletron*) Nell'Esercizio 7.19 avete scritto una simulazione software di un computer che esegue i programmi scritti nel Linguaggio Macchina Simpletron (LMS). In questo esercizio, proporremo diverse modifiche e miglioramenti al Simulatore Simpletron. Negli Esercizi 12.26 e 12.27, proporremo la costruzione di un compilatore che convertirà i programmi scritti in un linguaggio ad alto livello (una variante del BASIC) nel Linguaggio Macchina del Simpletron. Alcune delle seguenti modifiche e migliorie potranno essere necessarie per eseguire i programmi prodotti dal compilatore.

a) Estendete la memoria del Simulatore Simpletron, in modo che possa contenere 1000 locazioni e consentire al Simpletron di gestire programmi più corposi.

b) Fate in modo che il simulatore possa eseguire il calcolo del resto. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
c) Fate in modo che il simulatore possa calcolare l'elevamento a potenza. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
d) Modificate il simulatore in modo che utilizzi dei valori esadecimali, invece di quelli interi, per rappresentare le istruzioni del Linguaggio Macchina Simpletron.
e) Modificate il simulatore per consentire la visualizzazione di un carattere newline. Sarà necessario aggiungere un'istruzione al Linguaggio Macchina Simpletron.
f) Modificate il simulatore in modo che possa elaborare anche dei valori in virgola mobile oltre a quelli interi.
g) Modificate il simulatore in modo che possa gestire l'input di stringhe. [*Suggerimento*: ogni parola del Simpletron potrà essere suddivisa in due gruppi contenenti ognuno un intero di due cifre. Ogni intero di due cifre rappresenterà il valore decimale ASCII equivalente a un carattere. Aggiungete un'istruzione in linguaggio macchina che prenda in input una stringa e la immagazzini in una specifica locazione della memoria del Simpletron. In quella locazione, la prima metà della parola conterrà il numero di caratteri inclusi nella stringa (ovverosia, la sua lunghezza). Ogni mezza parola successiva conterrà un carattere ASCII espresso con un valore decimale di due cifre. L'istruzione in linguaggio macchina convertirà ogni carattere nel suo equivalente ASCII e lo assegnerà alla mezza parola.]
h) Modificate il simulatore in modo che gestisca l'output delle stringhe immagazzinate nel formato descritto nella parte (g). [*Suggerimento*: aggiungete al linguaggio macchina un'istruzione che visualizzi una stringa cominciando da una certa locazione della memoria del Simpletron. In quella locazione, la prima metà della parola rappresenterà la lunghezza della stringa misurata in caratteri. Ogni mezza parola successiva conterrà un carattere ASCII espresso con un valore decimale di due cifre. L'istruzione in linguaggio macchina controllerà la lunghezza e visualizzerà la stringa, traducendo ogni numero di due cifre nel carattere equivalente.]

7.30 Che cosa farà il seguente programma?

```
1  /* ex07_30.c */
2  /* Che cosa farà questo programma? */
3  #include <stdio.h>
4
5  int mystery3( const char *s1, const char *s2 ); /* prototipo */
6
7  int main()
8  {
9     char string1[ 80 ]; /* crea un vettore di caratteri */
10    char string2[ 80 ]; /* crea un vettore di caratteri */
11
12    printf( "Enter two strings: " );
13    scanf( "%s%s", string1, string2 );
14
15    printf( "The result is %d\n", mystery3( string1, string2 ) );
16
17    return 0; /* indica che il programma è terminato con successo */
18
19 } /* fine della funzione main */
20
21 int mystery3( const char *s1, const char *s2 )
22 {
23    for (  ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ ) {
24
```

```
25          if (*s1 != *s2) {
26             return 0;
27          } /* fine del comando if */
28
29       }
30
31       return 1;
32
33    } /* fine della funzione mystery3 */
```

Capitolo 8

I caratteri e le stringhe in C

Obiettivi

- Essere in grado di utilizzare le funzioni incluse nella libreria per la manipolazione dei caratteri (`ctype`).
- Essere in grado di utilizzare le funzioni per l'input/output delle stringhe e dei caratteri, incluse nella libreria per l'input/output standard (`stdio`).
- Essere in grado di utilizzare le funzioni per la conversione delle stringhe, incluse nella libreria di utilità standard (`stdlib`).
- Essere in grado di utilizzare le funzioni per l'elaborazione delle stringhe, incluse nella libreria per la manipolazione delle stringhe (`string`).
- Apprezzare la potenza insita nelle librerie di funzioni come un mezzo per ottenere la riusabilità del software.

8.1 Introduzione

In questo capitolo presenteremo le funzioni incluse nella libreria standard del C che semplificano l'elaborazione delle stringhe e dei caratteri. Tali funzioni consentiranno al programma di elaborare i caratteri, le stringhe, le righe di testo e i blocchi di memoria.

Il capitolo discuterà delle tecniche utilizzate per sviluppare gli editor di testo, i word processor, i software per l'impaginazione, i sistemi computerizzati per la composizione tipografica e altri tipi di software per l'elaborazione dei testi. Le manipolazioni del testo eseguite dalle funzioni per l'input/output formattato, come `printf` e `scanf`, potranno essere implementate utilizzando le funzioni discusse in questo capitolo.

8.2 I concetti fondamentali delle stringhe e dei caratteri

I caratteri sono i mattoncini fondamentali per la costruzione dei programmi sorgenti. Ogni programma è formato da una sequenza di caratteri che, qualora siano raggruppati in modo significativo, sarà interpretata dal computer come una serie di istruzioni da utilizzare per svolgere un compito. Un programma potrà contenere delle *costanti di carattere*. Queste sono dei valori `int` rappresentati da un carattere posto tra apici singoli. Il valore della costante di carattere corrisponde all'intero che gli sarà stato associato nell'insieme dei caratteri della macchina. Per esempio `'z'` rappresenta il valore intero di `z` e `'\n'` quello del carattere newline.

Una stringa è una serie di caratteri trattati come una singola unità. Una stringa potrà includere lettere, numeri e vari *caratteri speciali* come +, -, *, /, $ e tanti altri. Nel linguaggio C, le *stringhe letterali* o *costanti di stringa* si scrivono tra virgolette, come nei seguenti esempi:

"John Q. Doe"	(un nome)
"99999 Main Street"	(un indirizzo)
"Waltham, Massachusetts"	(una città e uno stato)
"(201) 555-1212"	(un numero telefonico)

Nel linguaggio C una stringa è in realtà un vettore di caratteri che termina con il *carattere nullo* (`'\0'`). Per accedere a una stringa si utilizzerà un puntatore che farà riferimento al suo primo carattere. Il valore di una stringa corrisponde all'indirizzo del suo primo carattere. Ne consegue che nel linguaggio C è corretto affermare che *una stringa è un puntatore*: in effetti, è un puntatore al primo carattere della stringa. In questo senso le stringhe sono come i vettori, poiché anche questi sono puntatori al loro primo elemento.

Una stringa potrà essere utilizzata in una dichiarazione per inizializzare un vettore di caratteri o una variabile di tipo `char *`. Le dichiarazioni

```
char color[] = "blue";
char *colorPtr = "blue";
```

inizializzeranno le rispettive variabili con la stringa `"blue"`. La prima dichiarazione creerà il vettore `color` di 5 elementi, contenente i caratteri `'b'`, `'l'`, `'u'`, `'e'` e `'\0'`. La seconda dichiarazione creerà la variabile di tipo puntatore `colorPtr` che farà riferimento alla stringa `"blue"` immagazzinata da qualche parte nella memoria.

Obiettivo portabilità 8.1

Alcuni compilatori potrebbero sistemare una variabile di tipo `char *` *che sia stata inizializzata con una stringa letterale, in una locazione di memoria in cui non possa essere modificata. Immagazzinate le stringhe letterali in vettori di caratteri, qualora abbiate bisogno di modificarle e vogliate assicurarvi di poterlo fare su tutti i sistemi.*

Avremmo potuto scrivere la dichiarazione del vettore precedente anche in questo modo

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Nel dichiarare un vettore di caratteri che debba contenere una stringa, sarà necessario assicurarsi che il vettore sia grande a sufficienza per immagazzinare la stringa e il carattere nullo di terminazione. La dichiarazione precedente determinerà automaticamente la dimensione del vettore basandosi sul numero degli elementi inclusi nella lista degli inizializzatori.

Errore tipico 8.1

Allocare in un vettore di caratteri uno spazio insufficiente a immagazzinare il carattere nullo di terminazione della stringa è un errore.

Errore tipico 8.2

Visualizzare una "stringa" che non contenga un carattere nullo di terminazione è un errore.

Collaudo e messa a punto 8.1

Assicuratevi che il vettore di caratteri in cui immagazzinerete le vostre stringhe sia dimensionato in modo tale da poter contenere quella più lunga. Il C non pone limiti alla

lunghezza delle stringhe immagazzinate. Nel caso che una stringa sia più lunga del vettore in cui dovrà essere immagazzinata, i caratteri in eccesso sostituiranno i dati memorizzati nelle locazioni di memoria successive al vettore.

Una stringa potrà essere assegnata a un vettore utilizzando la funzione `scanf`. Per esempio, la seguente istruzione assegnerà una stringa al vettore di caratteri `word[ 20 ]`:

```
scanf( "%s", word );
```

La stringa immessa dall'utente sarà immagazzinata in `word`. Osservate che `word` è un vettore, ovverosia un puntatore, e di conseguenza l'operatore `&` non sarà necessario con l'argomento `word`. La funzione `scanf` leggerà i caratteri finché non avrà incontrato uno spazio, un newline o l'indicatore di fine del file. Osservate che la lunghezza della stringa non potrà superare i 19 caratteri poiché si dovrà lasciare lo spazio per quello di terminazione. Affinché un vettore di caratteri possa essere visualizzato come una stringa, il vettore dovrà contenere un carattere nullo di terminazione.

Errore tipico 8.3

Trattare un carattere singolo come se fosse una stringa. Una stringa è un puntatore ed è quindi molto probabile che sia un intero considerevolmente grande. Un carattere, invece, è un intero piccolo (i valori dell'ASCII sono compresi tra 0 e 255). Su molti sistemi ciò provocherà una "violazione di accesso", perché gli indirizzi della memoria bassa sono riservati per degli scopi speciali, come la gestione degli interrupt del sistema operativo.

Errore tipico 8.4

Fornire un carattere come argomento di una funzione che invece attende una stringa è un errore di sintassi.

Errore tipico 8.5

Fornire una stringa come argomento di una funzione che invece attende un carattere è un errore di sintassi.

8.3 La libreria per la gestione dei caratteri

La libreria per la gestione dei caratteri include diverse funzioni che svolgono utili controlli e manipolazioni sui dati di tipo carattere. Ogni funzione riceve come argomento un carattere, rappresentato da un `int` o un `EOF`. Come abbiamo già affermato nel Capitolo 4, i caratteri sono gestiti spesso come se fossero degli interi perché in C un carattere corrisponde a un intero di 1 byte. `EOF` corrisponde normalmente al valore –1 e alcune architetture hardware non consentono l'immagazzinamento di valori negativi nelle variabili di tipo `char`; quindi le funzioni per la gestione dei caratteri trattano questi ultimi come degli interi. La Figura 8.1 riassume le funzioni incluse nella libreria per la gestione dei caratteri.

Collaudo e messa a punto 8.2

Dovrete includere il file di intestazione `<ctype.h>` *qualora utilizziate le funzioni contenute nella libreria per la gestione dei caratteri.*

Prototipo	Descrizione della funzione
`int isdigit( int c )`	Restituisce un valore vero se `c` è una cifra e 0 (falso) in caso contrario.
`int isalpha( int c )`	Restituisce un valore vero se `c` è una lettera e 0 in caso contrario.
`int isalnum( int c )`	Restituisce un valore vero se `c` è una cifra o una lettera e 0 in caso contrario.
`int isxdigit( int c )`	Restituisce un valore vero se `c` è una cifra esadecimale e 0 in caso contrario. (Consultate l'Appendice E, "I sistemi numerici", per una spiegazione dettagliata dei numeri binari, ottali, decimali ed esadecimali).
`int islower( int c )`	Restituisce un valore vero se `c` è una lettera minuscola e 0 in caso contrario.
`int isupper( int c )`	Restituisce un valore vero se `c` è una lettera maiuscola e 0 in caso contrario.
`int tolower( int c )`	Restituisce la lettera minuscola corrispondente, se `c` è una maiuscola, e l'argomento inalterato in caso contrario.
`int toupper( int c )`	Restituisce la lettera maiuscola corrispondente, se `c` è una minuscola, e l'argomento inalterato in caso contrario.
`int isspace( int c )`	Restituisce un valore vero, se `c` è un carattere di spazio bianco (newline (`'\n'`), spazio (`' '`), salto pagina (`'\f'`), ritorno carrello (`'\r'`), tabulazione orizzontale (`'\t'`) o tabulazione verticale (`'\v'`)) e 0 in caso contrario.
`int iscntrl( int c )`	Restituisce un valore vero se `c` è un carattere di controllo e 0 in caso contrario.
`int ispunct( int c )`	Restituisce un valore vero se `c` è un carattere stampabile diverso da uno spazio, una cifra o una lettera, e 0 in caso contrario.
`int isprint( int c )`	Restituisce un valore vero se `c` è un carattere stampabile, incluso lo spazio (`' '`), e 0 in caso contrario.
`int isgraph( int c )`	Restituisce un valore vero se `c` è un carattere stampabile diverso dallo spazio (`' '`) e 0 in caso contrario.

Figura 8.1 Funzioni incluse nella libreria per la gestione dei caratteri

Il programma della Figura 8.2 mostra l'utilizzo delle funzioni `isdigit`, `isalpha`, `isalnum` e `isxdigit`. La funzione `isdigit` determina se il suo argomento è un numero (`0-9`). La funzione `isalpha` determina se il suo argomento è una lettera maiuscola (`A-Z`) o minuscola (`a-z`). La funzione `isalnum` determina se il suo argomento è un numero, una lettera maiuscola o una minuscola. La funzione `isxdigit` determina se il suo argomento è una cifra esadecimale (`A-F`, `a-f`, `0-9`).

Il programma della Figura 8.2 utilizzerà l'operatore condizionale (`?:`) in combinazione con ognuna delle funzioni, per determinare se per ogni carattere controllato dovrà visualizzare la stringa `" is a "` o `" is not a "`. Per esempio, l'espressione

```
isdigit( '8' ) ? "8 is a " : "8 is not a "
```

indica che se `'8'` è una cifra e quindi se `isdigit` restituisce il valore vero (diverso da zero), sarà visualizzata la stringa `"8 is a "`, mentre se `'8'` non è una cifra e quindi se `isdigit` restituisce `0`, sarà visualizzata la stringa `"8 is not a "`.

```
1  /* Fig. 8.2: fig08_02.c
2     Usare le funzioni isdigit, isalpha, isalnum e isxdigit */
3  #include <stdio.h>
4  #include <ctype.h>
5
6  int main()
7  {
8     printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
9        isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
10       isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );
11
12    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
13       "According to isalpha:",
14       isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
15       isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
16       isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
17       isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );
18
19    printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
20       "According to isalnum:",
21       isalnum( 'A' ) ? "A is a " : "A is not a ",
22       "digit or a letter",
23       isalnum( '8' ) ? "8 is a " : "8 is not a ",
24       "digit or a letter",
25       isalnum( '#' ) ? "# is a " : "# is not a ",
26       "digit or a letter" );
27
28    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
29       "According to isxdigit:",
30       isxdigit( 'F' ) ? "F is a " : "F is not a ",
31       "hexadecimal digit",
32       isxdigit( 'J' ) ? "J is a " : "J is not a ",
33       "hexadecimal digit",
34       isxdigit( '7' ) ? "7 is a " : "7 is not a ",
35       "hexadecimal digit",
36       isxdigit( '$' ) ? "$ is a " : "$ is not a ",
37       "hexadecimal digit",
38       isxdigit( 'f' ) ? "f is a " : "f is not a ",
39       "hexadecimal digit" );
40
41    return 0; /* indica che il programma è terminato con successo */
42
43 } /* fine della funzione main */
```

Figura 8.2 Usare isdigit, isalpha, isalnum e isxdigit (continua)

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A ia a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Figura 8.2 Usare isdigit, isalpha, isalnum e isxdigit

Il programma della Figura 8.3 mostra l'utilizzo delle funzioni `islower`, `isupper`, `tolower` e `toupper`. La funzione `islower` determina se il suo argomento è una lettera minuscola (`a-z`). La funzione `isupper` determina se il suo argomento è una lettera maiuscola (`A-Z`). La funzione `tolower` converte una lettera maiuscola in una minuscola. Nel caso che l'argomento non sia una lettera maiuscola, `tolower` lo restituirebbe inalterato. La funzione `toupper` converte una lettera minuscola in una maiuscola. Nel caso che l'argomento non sia una lettera minuscola, `toupper` lo restituirebbe invariato.

```
1  /* Fig. 8.3: fig08_03.c
2     Usare le funzioni islower, isupper, tolower, toupper */
3  #include <stdio.h>
4  #include <ctype.h>
5
6  int main()
7  {
8     printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
9             "According to islower:",
10            islower( 'p' ) ? "p is a " : "p is not a ",
11            "lowercase letter",
12            islower( 'P' ) ? "P is a " : "P is not a ",
13            "lowercase letter",
14            islower( '5' ) ? "5 is a " : "5 is not a ",
15            "lowercase letter",
```

Figura 8.3 Usare le funzioni islower, isupper, tolower e toupper (continua)

```
16             islower( '!' ) ? "! is a " : "! is not a ",
17             "lowercase letter");
18
19      printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
20             "According to isupper:",
21             isupper( 'D' ) ? "D is an " : "D is not an ",
22             "uppercase letter",
23             isupper( 'd' ) ? "d is an " : "d is not an ",
24             "uppercase letter",
25             isupper( '8' ) ? "8 is an " : "8 is not an ",
26             "uppercase letter",
27             isupper( '$' ) ? "$ is an " : "$ is not an ",
28             "uppercase letter" );
29
30      printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
31             "u converted to uppercase is ", toupper( 'u' ),
32             "7 converted to uppercase is ", toupper( '7' ),
33             "$ converted to uppercase is ", toupper( '$' ),
34             "L converted to lowercase is ", tolower( 'L' ) );
35
36      return 0; /* indica che il programma è terminato con successo */
37
38   } /* fine della funzione main */
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

Figura 8.3 Usare le funzioni islower, isupper, tolower e toupper

La figura 8.4 mostra l'utilizzo delle funzioni `isspace`, `iscntrl`, `ispunct`, `isprint` e `isgraph`. La funzione `isspace` determina se il suo argomento è uno dei seguenti caratteri di spazio bianco: spazio (`' '`), salto pagina (`'\f'`), newline (`'\n'`), ritorno carrello (`'\r'`), tabulazione orizzontale (`'\t'`) o verticale (`'\v'`). La funzione `iscntrl` determina se il suo argomento è uno dei seguenti caratteri di controllo: tabulazione orizzontale (`'\t'`) o verticale (`'\v'`), allarme (`'\a'`), backspace (`'\b'`), ritorno carrello (`'\r'`) o newline (`'\n'`). La funzione `ispunct` determina se il suo argomento è un carattere stampabile diverso dallo

spazio, da una cifra o da una lettera come ad esempio $, #, (,), [,], {, }, ;, :, %, ecc. La funzione `isprint` determina se il suo argomento è un carattere che possa essere visualizzato sullo schermo (incluso lo spazio). La funzione `isgraph` controlla gli stessi caratteri di `isprint` escludendo lo spazio.

```
1  /* Fig. 8.4: fig08_04.c
2     Usare le funzioni isspace, iscntrl, ispunct, isprint, isgraph */
3  #include <stdio.h>
4  #include <ctype.h>
5
6  int main()
7  {
8     printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
9         "According to isspace:",
10        "Newline", isspace( '\n' ) ? " is a " : " is not a ",
11        "whitespace character", "Horizontal tab",
12        isspace( '\t' ) ? " is a " : " is not a ",
13        "whitespace character",
14        isspace( '%' ) ? "% is a " : "% is not a ",
15        "whitespace character" );
16
17    printf( "%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
18        "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
19       "control character", iscntrl( '$' ) ? "$ is a " : "$ is not a ",
20        "control character" );
21
22    printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
23       "According to ispunct:",
24        ispunct( ';' ) ? "; is a " : "; is not a ",
25        "punctuation character",
26        ispunct( 'Y' ) ? "Y is a " : "Y is not a ",
27        "punctuation character",
28        ispunct( '#' ) ? "# is a " : "# is not a ",
29        "punctuation character" );
30
31    printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
32        isprint( '$' ) ? "$ is a " : "$ is not a ",
33        "printing character",
34        "Alert", isprint( '\a' ) ? " is a " : " is not a ",
35        "printing character" );
36
37    printf( "%s\n%s%s\n%s%s%s\n",  "According to isgraph:",
38        isgraph( 'Q' ) ? "Q is a " : "Q is not a ",
39        "printing character other than a space",
40        "Space", isgraph( ' ' ) ? " is a " : " is not a ",
41        "printing character other than a space" );
42
43    return 0; /* indica che il programma è terminato con successo */
44
45 } /* fine della funzione main */
```

Figura 8.4 Usare isspace, iscntrl, ispunct, isprint e isgraph (continua)

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

Figura 8.4 Usare isspace, iscntrl, ispunct, isprint e isgraph

8.4 Le funzioni per la conversione delle stringhe

Questa sezione presenterà le *funzioni per la conversione delle stringhe* tratte dalla *libreria di utilità generiche* (`<stdlib.h>`). Queste funzioni convertono le stringhe formate da numeri in valori interi e in virgola mobile. La Figura 8.5 riassume le funzioni per la conversione delle stringhe. Osservate l'utilizzo di `const` per dichiarare la variabile `nPtr` nelle intestazioni di funzione (che vanno lette da destra a sinistra come "`nPtr` è un puntatore a una costante di carattere"); `const` dichiara che il valore dell'argomento non sarà modificato.

Prototipo di funzione	Descrizione della funzione
`double atof( const char *nPtr )`	Converte la stringa `nPtr` in un `double`.
`int atoi( const char *nPtr )`	Converte la stringa `nPtr` in un `int`.
`long atol( const char *nPtr )`	Converte la stringa `nPtr` in un `long int`.
`double strtod( const char *nPtr, char **endPtr )`	Converte la stringa `nPtr` in un `double`.
`long strtol( const char *nPtr, char **endPtr, int base )`	Converte la stringa `nPtr` in un `long`.
`unsigned long strtoul( const char *nPtr, char **endPtr, int base )`	Converte la stringa `nPtr` in un `unsigned long`.

Figura 8.5 Funzioni per la conversione delle stringhe incluse nella libreria di utilità generiche

Collaudo e messa a punto 8.3

Includete il file di intestazione `<stdlib.h>` *quando utilizzate le funzioni incluse nella libreria di utilità generiche.*

La funzione `atof` (Figura 8.6) restituisce il suo argomento (una stringa che rappresenta un numero in virgola mobile) convertito in un valore `double`. Nel caso che il valore convertito non possa essere rappresentato (per esempio, qualora il primo carattere della stringa non sia un numero) il comportamento della funzione `atof` sarà indefinito.

```
1  /* Fig. 8.6: fig08_06.c
2     Usare atof */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     double d; /* variabile per memorizzare la stringa convertita */
9
10    d = atof( "99.0" );
11
12    printf( "%s%.3f\n%s%.3f\n",
13           "The string \"99.0\" converted to double is ", d,
14           "The converted value divided by 2 is ",
15           d / 2.0 );
16
17    return 0; /* indica che il programma è terminato con successo */
18
19 } /* fine della funzione main */
```

```
The string "99.0" converted to double is 99.000
The converted value divided by 2 is 49.500
```

Figura 8.6 Usare atof

La funzione `atoi` (Figura 8.7) restituisce il suo argomento (una stringa di cifre che rappresenta un intero) convertito in un valore `int`. Nel caso che il valore convertito non possa essere rappresentato il comportamento della funzione `atoi` sarà indefinito.

```
1  /* Fig. 8.7: fig08_07.c
2     Usare atoi */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     int i; /* variabile per memorizzare la stringa convertita */
9
```

Figura 8.7 Usare atoi (continua)

```
10      i = atoi( "2593" );
11
12      printf( "%s%d\n%s%d\n",
13              "The string \"2593\" converted to int is ", i,
14              "The converted value minus 593 is ", i - 593 );
15
16      return 0; /* indica che il programma è terminato con successo */
17
18   } /* fine della funzione main */
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Figura 8.7 Usare atoi

La funzione `atol` (Figura 8.8) restituisce il suo argomento (una stringa di cifre che rappresenta un numero intero grande) convertito in un valore `long`. Nel caso che il valore convertito non possa essere rappresentato il comportamento di `atol` sarà indefinito. Le funzioni `atoi` e `atol` funzioneranno in modo identico qualora gli `int` e i `long` siano entrambi immagazzinati in 4 byte.

```
1   /* Fig. 8.8: fig08_08.c
2      Usare atol */
3   #include <stdio.h>
4   #include <stdlib.h>
5
6   int main()
7   {
8      long l; /* variabile per memorizzare la stringa convertita */
9
10     l = atol( "1000000" );
11
12     printf( "%s%ld\n%s%ld\n",
13         "The string \"1000000\" converted to long int is ", l,
14         "The converted value divided by 2 is ", l / 2 );
15
16     return 0; /* indica che il programma è terminato con successo */
17
18  } /* fine della funzione main */
```

```
The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000
```

Figura 8.8 Usare atol

La funzione `strtod` (Figura 8.9) converte in un valore `double` una sequenza di caratteri che rappresenta un numero in virgola mobile. La funzione riceverà due argomenti: una stringa (`char *`) e un puntatore a una stringa (`char **`). La stringa conterrà la sequenza di

caratteri da convertire in `double`. Al puntatore sarà assegnata la locazione del primo carattere successivo alla porzione della stringa che sarà stata convertita. La riga 14

```
d = strtod( string, &stringPtr );
```

indica che a `d` sarà assegnato il valore `double` convertito da `string`, mentre a `stringPtr` sarà assegnata la locazione del primo carattere successivo al valore convertito (`51,2`) da `string`.

```
1  /* Fig. 8.9: fig08_09.c
2     Usare strtod */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     /* inizializza il puntatore a stringa */
9     const char *string = "51.2% are admitted"; /* inizializza
                                                    la stringa */
10
11    double d; /* variabile per memorizzare la sequenza convertita */
12    char *stringPtr; /* crea un puntatore a carattere */
13
14    d = strtod( string, &stringPtr );
15
16    printf( "The string \"%s\" is converted to the\n", string);
17    printf( "double value %.2f and the string \"%s\"\n", d, stringPtr);
18
19    return 0; /* indica che il programma è terminato con successo */
20
21 } /* fine della funzione main */
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Figura 8.9 Usare strtod

La funzione `strtol` (Figura 8.10) converte in un `long` una sequenza di caratteri che rappresenta un intero. La funzione riceverà tre argomenti: una stringa (`char *`), un puntatore a una stringa e un intero. La stringa conterrà la sequenza di caratteri da convertire. Al puntatore sarà assegnata la locazione del primo carattere successivo alla porzione della stringa che sarà stata convertita. L'intero indicherà la *base* del valore da convertire. L'istruzione

```
x = strtol( string, &remainderPtr, 0 );
```

nella Figura 8.10 indica che a `x` sarà assegnato il valore `long` convertito da `string`. Al secondo argomento, `remainderPtr`, sarà assegnata la parte rimanente di `string` dopo la conversione. Utilizzare `NULL` come secondo argomento farà in modo che il resto della stringa sia ignorato. Il terzo argomento, `0`, indica che il valore da convertire potrà essere in un formato ottale (base 8), decimale (base 10) o esadecimale (base 16). La base potrà essere specificata come 0 o qualsiasi altro valore compreso tra 2 e 36. Consultate l'Appendice E, I sistemi

numerici, per una spiegazione dettagliata dei sistemi numerici ottale, decimale ed esadecimale. Le rappresentazioni numeriche degli interi nelle basi comprese tra 11 e 36 utilizzano i caratteri A-Z per rappresentare i valori compresi tra 10 e 35. Per esempio, i valori esadecimali possono essere formati dalle cifre 0-9 e dai caratteri A-F. Un intero in base 11 può essere formato dalle cifre 0-9 e dal carattere A. Un intero in base 24 può essere formato dalle cifre 0-9 e dai caratteri A-N. Un intero in base 36 può essere formato dalle cifre 0-9 e dai caratteri A-Z.

```
1  /* Fig. 8.10: fig08_10.c
2     Usare strtol */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     const char *string = "-1234567abc"; /* inizializza il puntatore
                                              a stringa */
9
10    char *remainderPtr; /* crea un puntatore a carattere */
11    long x;             /* variabile per memorizzare la sequenza
                             convertita */
12
13    x = strtol( string, &remainderPtr, 0 );
14
15    printf( "%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
16        "The original string is ", string,
17        "The converted value is ", x,
18        "The remainder of the original string is ",
19        remainderPtr,
20        "The converted value plus 567 is ", x + 567 );
21
22    return 0; /* indica che il programma è terminato con successo */
23
24 } /* fine della funzione main */
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

Figura 8.10 Usare strtol

La funzione `strtoul` (Figura 8.11) converte in un `unsigned long` una sequenza di caratteri che rappresenta un intero `unsigned long`. La funzione agisce in modo identico a quello di `strtol`. L'istruzione

```
x = strtoul( string, &remainderPtr, 0 );
```

nella Figura 8.11 indica che a `x` sarà assegnato il valore `unsigned long` convertito da `string`. Al secondo argomento, `&remainderPtr`, sarà assegnata la parte rimanente di `string` dopo la

conversione. Il terzo argomento, `0`, indica che il valore da convertire potrà essere in un formato ottale, decimale o esadecimale.

```
1  /* Fig. 8.11: fig08_11.c
2     Usare strtoul */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8     const char *string = "1234567abc"; /* inizializza il puntatore
                                            a stringa */
9     unsigned long x; /* variabile per memorizzare la sequenza
                          convertita */
10    char *remainderPtr; /* crea un puntatore a carattere */
11
12    x = strtoul( string, &remainderPtr, 0 );
13
14    printf( "%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
15           "The original string is ", string,
16           "The converted value is ", x,
17           "The remainder of the original string is ",
18           remainderPtr,
19           "The converted value minus 567 is ", x - 567 );
20
21    return 0; /* indica che il programma è terminato con successo */
22
23 } /* fine della funzione main */
```

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

Figura 8.11 Usare strtoul

8.5 Le funzioni della libreria per l'input/output standard

Questa sezione presenterà diverse funzioni tratte dalla libreria per l'input/output standard (`<stdio.h>`) e, in particolar modo, quelle per la manipolazione dei dati di tipo carattere e stringa. La Figura 8.12 riassume le funzioni per l'input/output delle stringhe e dei caratteri incluse nella libreria per l'input/output standard.

Buona abitudine 8.4

Includete il file di intestazione `<stdio.h>` *quando utilizzate le funzioni incluse nella libreria per l'input/output standard.*

Prototipo di funzione	Descrizione della funzione
`int getchar( void )`	Legge il prossimo carattere dallo standard input e lo restituisce come valore intero.
`char *gets( char *s )`	Legge i caratteri dallo standard input e li immagazzina nel vettore `s`, finché non incontra un carattere newline o un indicatore di fine del file. Alla fine del vettore sarà accodato un carattere nullo di terminazione.
`int putchar( int c )`	Visualizza il carattere immagazzinato in `c`.
`int puts( const char *s )`	Visualizza la stringa seguita da un carattere di newline.
`int sprintf( char *s, const char *format, ... )`	È equivalente a `printf` eccetto che l'output non sarà visualizzato sullo schermo ma sarà immagazzinato nel vettore `s`.
`int sscanf( char *s, const char *format, ... )`	È equivalente a `scanf` eccetto che l'input non sarà letto dalla tastiera ma dal vettore `s`.

Figura 8.12 Le funzioni per i caratteri e le stringhe incluse nella libreria per l'input/output standard

Il programma della Figura 8.13 utilizzerà le funzioni `gets` e `putchar` per leggere una riga di testo dallo standard input (la tastiera) e inviare in output in ordine inverso i singoli caratteri della riga, usando una funzione ricorsiva. La funzione `gets` leggerà i caratteri dallo standard input e li immagazzinerà nel suo argomento, un vettore di tipo `char`, finché non avrà incontrato un carattere newline o un indicatore di fine del file. Un carattere nullo (`'\0'`) sarà accodato al vettore quando la lettura sarà stata completata. La funzione `putchar` visualizzerà il suo argomento di tipo carattere. Il programma invocherà la funzione ricorsiva `reverse` per visualizzare la riga di testo in ordine inverso. Nel caso che il primo carattere del vettore ricevuto da `reverse` sia il carattere nullo `'\0'`, `reverse` restituirà il controllo alla funzione chiamante. In caso contrario, `reverse` sarà invocata nuovamente con l'indirizzo del sottovettore che incomincia all'elemento `s[ 1 ]`, mentre il carattere `s[ 0 ]` sarà inviato in output con `putchar` quando la chiamata ricorsiva sarà stata completata. L'ordine delle due istruzioni nella porzione `else` del comando `if` costringerà `reverse` a raggiungere il carattere nullo di terminazione della stringa, prima che possa stampare un qualsiasi carattere. Nel momento in cui saranno state completate le chiamate ricorsive, i caratteri saranno inviati in output in ordine inverso.

```
1  /* Fig. 8.13: fig08_13.c
2     Usare gets e putchar */
3  #include <stdio.h>
4
5  void reverse( const char * const sPtr ); /* prototipo */
6
7  int main()
```

Figura 8.13 Usare gets e putchar (continua)

```
 8  {
 9     char sentence[ 80 ]; /* crea un vettore di caratteri */
10
11     printf( "Enter a line of text:\n" );
12
13     /* usa gets per leggere una linea di testo */
14     gets( sentence );
15
16     printf( "\nThe line printed backwards is:\n" );
17     reverse( sentence );
18
19     return 0; /* indica che il programma è terminato con successo */
20
21  } /* fine della funzione main */
22
23  /* i caratteri della stringa vengono visualizzati ricorsivamente
       in ordine inverso */
24  void reverse( const char * const sPtr )
25  {
26     /* se viene raggiunta la fine della stringa */
27     if ( sPtr[ 0 ] == '\0') { /* caso base */
28        return;
29     } /* fine del ramo if */
30     else { /* se la fine della stringa non è stata raggiunta */
31        reverse( &sPtr[ 1 ] ); /* passo risorsivo */
32
33        putchar( sPtr[ 0 ] ); /* usa putchar per visualizzare
                                   il carattere */
34     } /* fine del ramo else */
35
36  } /* fine della funzione reverse */
```

```
Enter a line of text;
Characters and Strings

The line printed backwards is:
sgnirtS dna sretcarahC
```

```
Enter a line of text:
able was I ere I saw elba

The line printed backwards is:
able was I ere I saw elba
```

Figura 8.13 Usare gets e putchar

Il programma della Figura 8.14 utilizzerà le funzioni `getchar` e `puts` per leggere i caratteri dallo standard input, immagazzinarli nel vettore `sentence` e visualizzarlo come se fosse una stringa. La funzione `getchar` leggerà un carattere dallo standard input e lo restituirà

come un valore intero. La funzione puts riceverà come argomento una stringa (char *) e la visualizzerà facendola seguire da un carattere di newline.

Il programma smetterà di prendere in input i caratteri quando getchar avrà letto il newline immesso dall'utente alla fine della riga di testo. Un carattere nullo sarà accodato al vettore sentence (riga 19) in modo che possa essere trattato come una stringa. Quindi la funzione puts visualizzerà la stringa contenuta in sentence.

```
1  /* Fig. 8.14: fig08_14.c
2     Usare getchar and puts */
3  #include <stdio.h>
4
5  int main()
6  {
7     char c; /* variabile per memorizzare il carattere inserito
                  dall'utente */
8     char sentence[ 80 ] ; /* crea un vettore di caratteri */
9     int i = 0; /* inizializza il contatore i */
10
11    /* sollecita l'utente a inserire una linea di testo */
12    puts( "Enter a line of text:" );
13
14    /* usa getchar per leggere ogni singolo carattere */
15    while ( ( c = getchar() ) != '\n' ) {
16       sentence[ i++ ] = c;
17    } /* fine del comando while */
18
19    sentence[ i ] = '\0';  /* termina la stringa */
20
21    /* usa puts per visualizzare sentence */
22    puts( "\nThe line entered was:" );
23    puts( sentence );
24
25    return 0; /* indica che il programma è terminato con successo */
26
27 } /* fine della funzione main */
```

```
Enter a line of text:
This is a test.

The line entered was:
This is a test.
```

Figura 8.14 Usare getchar e puts

Il programma della Figura 8.15 utilizzerà la funzione **sprintf** per riversare nel vettore di caratteri **s** i dati formattati. La funzione utilizza le stesse specifiche di conversione di **printf** (consultate il Capitolo 9, per una discussione dettagliata di tutte le potenzialità per la formattazione della stampa). Il programma riceverà in input un **int** e un **double** che dovranno essere formattati e stampati nel vettore **s**. Quest'ultimo sarà il primo argomento di **sprintf**.

```
1  /* Fig. 8.15: fig08_15.c
2     Usare sprintf */
3  #include <stdio.h>
4
5  int main()
6  {
7     char s[ 80 ]; /* crea un vettore di caratteri */
8     int x; /* valore da leggere */
9     double y; /* valore da leggere */
10
11    printf( "Enter an integer and a float:\n" );
12    scanf( "%d%f", &x, &y );
13
14    sprintf( s, "Integer:%6d\nFloat:%8.2f", x, y );
15
16    printf( "%s\n%s\n",
17           "The formatted output stored in array s is:", s );
18
19    return 0; /* indica che il programma è terminato con successo */
20
21 } /* fine della funzione main */
```

```
Enter an integer and a float:
298 87.375
The formatted output stored in array s is:
Integer:   298
Float:   87.38
```

Figura 8.15 Usare sprintf

Il programma della Figura 8.16 utilizzerà la funzione `sscanf` per leggere dal vettore di caratteri `s` dei dati formattati. La funzione utilizza le stesse specifiche di conversione di `scanf`. Il programma leggerà dal vettore `s` un `int` e un `double` e immagazzinerà i loro valori rispettivamente in `x` e `y`. In seguito i valori di `x` e `y` saranno visualizzati. Il vettore `s` sarà il primo argomento di `sscanf`.

```
1  /* Fig. 8.16: fig08_16.c
2     Usare sscanf */
3  #include <stdio.h>
4
5  int main()
6  {
7     char s[] = "31298 87.375"; /* inizializza il vettore s */
8     int x; /* valore da leggere */
9     double y; /* valore da leggere */
10
11    sscanf( s, "%d%lf", &x, &y );
12
```

Figura 8.16 Usare sscanf (continua)

```
13      printf( "%s\n%s%6d\n%s%8.3f\n",
14              "The values stored in character array s are:",
15              "integer:", x, "double:", y );
16
17      return 0; /* indica che il programma è terminato con successo */
18
19   } /* fine della funzione main */
```

```
The values stored in character array s are
integer:  31298
double:   87.375
```

Figura 8.16 Usare sscanf

8.6 Le funzioni per la manipolazione delle stringhe incluse nella libreria per la gestione delle stringhe

La libreria per la gestione delle stringhe (`<string.h>`) fornisce molte funzioni utili per manipolare le stringhe (*copiare* e *concatenare stringhe*), *confrontarle*, *ricercare* in esse dei caratteri o altre stringhe, *suddividerle in token* (separarle in componenti logici) e *determinarne la lunghezza*. Questa sezione presenterà le funzioni per la manipolazione delle stringhe tratte dalla libreria per la gestione delle stringhe. Le funzioni sono riassunte nella Figura 8.17. Tutte le funzioni, tranne `strncpy`, aggiungono il carattere nullo alla fine del loro risultato.

Prototipo di funzione	Descrizione della funzione
`char *strcpy( char *s1, const char *s2 )`	Copia la stringa `s2` nel vettore `s1`. Restituisce il valore di `s1`.
`char *strncpy( char *s1, const char *s2, size_t n )`	Copia un massimo di `n` caratteri dalla stringa `s2` nel vettore `s1`. Restituisce il valore di `s1`.
`char *strcat( char *s1, const char *s2 )`	Accoda la stringa `s2` al vettore `s1`. Il primo carattere di `s2` si sostituisce al carattere nullo di terminazione di `s1`. Restituisce il valore di `s1`.
`char *strncat( char *s1, const char *s2, size_t n )`	Accoda un massimo di `n` caratteri dalla stringa `s2` al vettore `s1`. Il primo carattere di `s2` si sostituisce al carattere nullo di terminazione di `s1`. Restituisce il valore di `s1`.

Figura 8.17 Le funzioni per la manipolazione delle stringhe incluse nella libreria per la gestione delle stringhe

Le funzioni `strncpy` e `strncat` specificano un parametro di tipo `size_t`, che viene definito dallo Standard C come il tipo intero del valore restituito dall'operatore `sizeof`.

Obiettivo portabilità 8.2

Il tipo `size_t` *è un sinonimo, dipendente dalla macchina, per* `unsigned long` *o* `unsigned int`.

Collaudo e messa a punto 8.5

Includete il file di intestazione `<string.h>` *quando utilizzate le funzioni incluse nella libreria per la gestione delle stringhe.*

La funzione `strcpy` copia il suo secondo argomento (una stringa) nel primo, ovverosia in un vettore di caratteri sufficientemente grande per immagazzinare la stringa e il relativo carattere nullo di terminazione. La funzione `strncpy` è equivalente a `strcpy`, eccetto che `strncpy` specifica il numero di caratteri che dovranno essere ricopiati dalla stringa nel vettore. Osservate che la funzione `strncpy` non ricopierà necessariamente il carattere nullo di terminazione del suo secondo argomento. Il carattere nullo di terminazione sarà ricopiato, soltanto se il numero dei caratteri da ricopiare sarà superiore alla lunghezza della stringa di almeno una unità. Per esempio, qualora il secondo argomento fosse `"test"`, il carattere nullo sarebbe ricopiato soltanto se il terzo argomento fosse almeno `5` (4 caratteri in `"test"` più un carattere nullo di terminazione). Qualora il terzo argomento fosse stato maggiore di `5`, al vettore sarebbero stati accodati dei caratteri nulli fino al raggiungimento del numero totale di caratteri specificato dal terzo argomento.

Errore tipico 8.6

Non accodare un carattere nullo di terminazione al primo argomento di una `strncpy`*, qualora il terzo argomento sia minore o uguale alla lunghezza della stringa indicata dal secondo argomento.*

Il programma della Figura 8.18 utilizzerà `strcpy` per ricopiare l'intera stringa `x` nel vettore `y`, mentre utilizzerà `strncpy` per ricopiare i primi 14 caratteri della stringa `x` nel vettore `z`. Al vettore `z` sarà accodato un carattere nullo (`'\0'`) perché l'invocazione di `strncpy` nel programma non scriverà un carattere nullo di terminazione, dato che il terzo argomento è inferiore alla lunghezza della stringa indicata dal secondo argomento.

La funzione `strcat` *accoda* il suo secondo argomento (una stringa) al primo, ovverosia a un vettore di caratteri che contiene una stringa. Il primo carattere del secondo argomento sostituirà il carattere nullo (`'\0'`) che termina la stringa del primo argomento. Il programmatore dovrà assicurarsi che il vettore utilizzato per immagazzinare la prima stringa sia sufficientemente grande per immagazzinare la prima stringa, la seconda e il carattere nullo di terminazione che sarà ricopiato dalla seconda stringa. La funzione `strncat` accoda un numero specificato di caratteri dalla seconda stringa alla prima. Un carattere nullo di terminazione sarà accodato automaticamente al risultato. Il programma della Figura 8.19 mostra l'utilizzo delle funzioni `strcat` e `strncat`.

```
1  /* Fig. 8.18: fig08_18.c
2     Usare strcpy e strncpy */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
```

Figura 8.18 Usare strcpy e strncpy (continua)

```
{
   char x[] = "Happy Birthday to You"; /* inizializza il vettore
                                          di caratteri x */
   char y[ 25 ]; /* crea il vettore di caratteri y */
   char z[ 15 ]; /* crea il vettore i caratteri z */

   /* copia il contenuto di x in y */
   printf( "%s%s\n%s%s\n",
          "The string in array x is: ", x,
          "The string in array y is: ", strcpy( y, x ) );

   /* copia i primi 14 caratteri di x in z.
      Non copia il carattere nullo */
   strncpy( z, x, 14 );

   z[ 14 ] = '\0';
   printf( "The string in array z is: %s\n", z );

   return 0; /* indica che il programma è terminato con successo */

} /* fine della funzione main */
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Figura 8.18 Usare strcpy e strncpy

```
/* Fig. 8.19: fig08_19.c
   Usare strcat e strncat */
#include <stdio.h>
#include <string.h>

int main()
{
   char s1[ 20 ] = "Happy "; /* inizializza il vettore
                                di caratteri s1 */
   char s2[] = "New Year "; /* inizializza il vettore di caratteri s2 */
   char s3[ 40 ] = ""; /* inizializza il vettore di caratteri s3
                          con la stringa vuota */

   printf( "s1 = %s\ns2 = %s\n", s1, s2 );

   /* accoda s2 a s1 */
   printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );

   /* accoda i primi 6 caratteri di s1 a s3.
      Inserisce '\0' dopo l'ultimo carattere */
   printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );

   /* accoda s1 a s3 */
```

Figura 8.19 Usare strcat e strncat (continua)

```
22      printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );
23
24      return 0; /* indica che il programma è terminato con successo */
25
26   } /* fine della funzione main */
```

```
s1 = Happy
s2 = New Year
strcat(s1, s2) = Happy New Year
strncat(s3, s1, 6) = Happy
strcat(s3, s1) = Happy Happy New Year
```

Figura 8.19 Usare strcat e strncat

8.7 Le funzioni di confronto incluse nella libreria per la gestione delle stringhe

Questa sezione presenterà le funzioni per il confronto delle stringhe, `strcmp` e `strncmp`, incluse nella libreria per la gestione delle stringhe. La Figura 8.20 illustra i prototipi e una breve descrizione di entrambe le funzioni.

Il programma della Figura 8.21 confronterà tre stringhe utilizzando le funzioni `strcmp` e `strncmp`. La funzione `strcmp` confronta il suo primo argomento con il secondo, un carattere per volta. La funzione restituirà 0 qualora le stringhe siano uguali, un valore negativo qualora la prima sia minore della seconda, e un valore positivo qualora la prima sia maggiore della seconda. La funzione `strncmp` è equivalente a `strcmp` eccetto che `strncmp` confronta un numero di caratteri specificato. La funzione `strncmp` non eseguirà il confronto con i caratteri successivi al `NULL` in una stringa. Il programma visualizzerà il valore intero restituito da ogni chiamata di funzione.

Errore tipico 8.7

Presumere che `strcmp` *e* `strncmp` *restituiscano il valore 1 qualora i loro argomenti siano uguali è un errore logico. In caso di uguaglianza entrambe le funzioni restituiscono 0 (stranamente l'equivalente del valore falso in C). Di conseguenza, quando verificate l'uguaglianza di due stringhe per determinare se siano uguali, il risultato di* `strcmp` *o di* `strncmp` *dovrà essere confrontato con 0.*

Prototipo di funzione	Descrizione della funzione
`int strcmp( const char *s1, const char *s2 )`	Confronta le stringhe `s1` e `s2`. La funzione restituirà 0, un valore minore di 0 o maggiore di 0 qualora `s1` sia rispettivamente uguale, minore o maggiore di `s2`.
`int strncmp( const char *s1, const char *s2, size_t n )`	Confronta un massimo di `n` caratteri di `s1` con la stringa `s2`. La funzione restituirà 0, un valore minore di 0 o maggiore di 0 qualora `s1` sia rispettivamente uguale, minore o maggiore di `s2`.

Figura 8.20 Le funzioni di confronto delle stringhe incluse nella libreria per la gestione delle stringhe

```
1  /* Fig. 8.21: fig08_21.c
2     Usare strcmp e strncmp */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     const char *s1 = "Happy New Year"; /* inizializza un puntatore
                                            a carattere */
9     const char *s2 = "Happy New Year"; /* inizializza un puntatore
                                            a carattere */
10    const char *s3 = "Happy Holidays"; /* inizializza un puntatore
                                            a carattere */
11
12    printf( "%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
13           "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
14           "strcmp(s1, s2) = ", strcmp( s1, s2 ),
15           "strcmp(s1, s3) = ", strcmp( s1, s3 ),
16           "strcmp(s3, s1) = ", strcmp( s3, s1 ) );
17
18    printf( "%s%2d\n%s%2d\n%s%2d\n",
19           "strncmp(s1, s3, 6) = ", strncmp( s1, s3, 6 ),
20           "strncmp(s1, s3, 7) = ", strncmp( s1, s3, 7 ),
21           "strncmp(s3, s1, 7) = ", strncmp( s3, s1, 7 ) );
22
23    return 0; /* indica che il programma è terminato con successo */
24
25 } /* fine della funzione main */
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Figura 8.21 Usare strcmp e strncmp

Per comprendere esattamente cosa significa per una stringa essere "maggiore di" o "minore di" un'altra stringa, considerate il processo di sistemazione in ordine alfabetico di una serie di cognomi. Senza dubbio, il lettore sistemerebbe "Jones" prima di "Smith", poiché la prima lettera di "Jones" precede nell'alfabeto l'iniziale di "Smith". Ma l'alfabeto è più di un semplice elenco di 26 lettere: è un elenco ordinato di caratteri. Ogni lettera occupa una posizione specifica all'interno dell'elenco. "Z" non è semplicemente una lettera dell'alfabeto: "Z" è la ventiseiesima lettera dell'alfabeto.

Ma in che modo il computer potrà sapere se una particolare lettera precede un'altra? Tutti i caratteri sono rappresentati all'interno del computer con dei *codici numerici*; nel momento in cui il computer confronta due stringhe esso confronta in realtà i codici numerici dei caratteri inclusi nelle stringhe.

Obiettivo portabilità 8.3

I codici numerici interni usati per rappresentare i caratteri potrebbero essere diversi sui vari computer.

Nel tentativo di standardizzare le rappresentazioni dei caratteri, la maggior parte dei produttori di computer ha progettato le proprie macchine in modo da utilizzare uno dei due schemi di codifica più diffusi: l'*ASCII o* l'*EBCDIC*. ASCII è l'acronimo di "American Standard Code for Information Interchange" (Codice Standard Americano per lo Scambio delle Informazioni). EBCDIC è l'acronimo di "Extended Binary Coded Decimal Interchange Code" (Codice di Scambio Decimale Codificato in Binario Esteso). Esistono altri schemi di codifica, ma questi due sono i più diffusi. Il recente Standard Unicode illustra una specifica per codificare in modo consistente una vasta maggioranza dei caratteri e simboli in uso nel mondo. Per apprendere ulteriori dettagli sullo Standard Unicode visitate il sito `www.unicode.org`.

L'ASCII, l'EBCDIC e l'Unicode sono chiamati *insiemi di caratteri*. Le manipolazioni delle stringhe e dei caratteri comportano in realtà la manipolazione degli appropriati codici numerici invece dei caratteri. Ciò spiega l'interscambiabilità dei caratteri e dei piccoli interi nel C. Dato che ha un senso affermare che un codice numerico è maggiore, minore o uguale a un altro codice numerico, diventa possibile mettere in relazione vari caratteri o stringhe gli uni con gli altri, facendo riferimento ai codici dei caratteri. L'Appendice D contiene un elenco dei codici utilizzati per i caratteri ASCII.

8.8 Le funzioni di ricerca incluse nella libreria per la gestione delle stringhe

Questa sezione presenterà le funzioni incluse nella libreria per la gestione delle stringhe, utilizzate per eseguire le ricerche di caratteri o di stringhe all'interno delle stringhe. Le funzioni sono riassunte nella Figura 8.22. Osservate che le funzioni `strcspn` e `strspn` restituiscono un valore di tipo `size_t`.

Prototipo di funzione	Descrizione della funzione
`char *strchr( const char *s, int c )`	Individua la prima occorrenza del carattere `c` nella stringa `s`. Qualora `c` sia stato trovato, sarà restituito un puntatore alla locazione di `c` in `s`. Altrimenti sarà restituito un puntatore `NULL`.
`size_t strcspn( const char *s1, const char *s2 )`	Determina e restituisce la lunghezza del segmento iniziale della stringa `s1` che non contenga nessuno dei caratteri inclusi in `s2`.

Figura 8.22 Le funzioni di ricerca contenute nella libreria per la gestione delle stringhe (continua)

Prototipo di funzione	Descrizione della funzione
`size_t strspn( const char *s1, const char *s2 )`	Determina e restituisce la lunghezza del segmento iniziale della stringa `s1` che contenga soltanto i caratteri inclusi in `s2`.
`char *strpbrk( const char *s1, const char *s2 )`	Individua nella stringa `s1` la prima occorrenza di uno qualsiasi dei caratteri inclusi in `s2`. Qualora un carattere della stringa `s2` sia stato ritrovato in `s1`, sarà restituito un puntatore alla locazione di quel carattere nella stringa `s1`. Altrimenti sarà restituito un puntatore `NULL`.
`char *strrchr( const char *s, int c )`	Individua l'ultima occorrenza di `c` nella stringa `s`. Qualora `c` sia stato ritrovato, sarà restituito un puntatore alla locazione di `c` nella stringa `s`. In caso contrario sarà restituito un puntatore `NULL`.
`char *strstr( const char *s1, const char *s2 )`	Individua la prima occorrenza della stringa `s2` in `s1`. Qualora la stringa sia stata ritrovata, sarà restituito un puntatore alla locazione di `s2` nella stringa `s1`. In caso contrario sarà restituito un puntatore `NULL`.
`char *strtok( char *s1, const char *s2 )`	Una sequenza di invocazioni di `strtok` suddividerà la stringa `s1` in "token" (componenti logici come, per esempio, le parole in una riga di testo) separati dai caratteri contenuti nella stringa `s2`. La prima invocazione dovrà contenere `s1` come primo argomento, mentre quelle susseguenti dovranno contenere `NULL`, qualora si desideri continuare a suddividere la stessa stringa. Ogni invocazione restituirà un puntatore al token corrente. Nel caso non ci fossero più token la funzione restituirà un `NULL`.

Figura 8.22 Le funzioni di ricerca contenute nella libreria per la gestione delle stringhe

La funzione `strchr` ricerca in una stringa la prima occorrenza di un carattere. Nel caso che questo sia stato ritrovato `strchr` restituirà un puntatore alla locazione di quel carattere nella stringa, altrimenti `strchr` restituirà un puntatore `NULL`. Il Programma della Figura 8.23 utilizzerà `strchr` per ricercare la prima occorrenza di `'a'` e `'z'` nella stringa `"This is a test"`.

```
1  /* Fig. 8.23: fig08_23.c
2     Usare strchr */
3  #include <stdio.h>
4  #include <string.h>
5
```

Figura 8.23 Usare strchr (continua)

```
6   int main()
7   {
8     const char *string = "This is a test"; /* inizializza un puntatore
                                                 a carattere */
9     char character1 = 'a'; /* inizializza character1 */
10    char character2 = 'z'; /* inizializza character2 */
11
12    /* se character1 viene trovato all'interno di string */
13    if ( strchr( string, character1 ) != NULL ) {
14       printf( "\'%c\' was found in \"%s\".\n",
15             character1, string );
16    } /* fine del ramo if */
17    else {
18       printf( "\'%c\' was not found in \"%s\".\n",
19             character1, string );
20    } /* fine del ramo else */
21
22    /* se character2 viene trovato all'interno di string */
23    if ( strchr( string, character2 ) != NULL ) {
24       printf( "\'%c\' was found in \"%s\".\n",
25             character2, string );
26    } /* fine del ramo if */
27    else {
28       printf( "\'%c\' was not found in \"%s\".\n",
29             character2, string );
30    } /* fine del ramo else */
31
32    return 0; /* indica che il programma è terminato con successo */
33
34  } /* fine della funzione main */
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

Figura 8.23 Usare strchr

La funzione `strcspn` (Figura 8.24) determina la lunghezza del segmento iniziale della stringa indicata dal suo primo argomento, che non contenga nessuno dei caratteri inclusi in quella indicata dal suo secondo argomento. La funzione restituirà la lunghezza del suddetto segmento.

```
1   /* Fig. 8.24: fig08_24.c
2      Usare strcspn */
3   #include <stdio.h>
4   #include <string.h>
5
6   int main()
7   {
8      /* inizializza due puntatori a carattere */
```

Figura 8.24 Usare strcspn (continua)

```
 9      const char *string1 = "The value is 3.14159";
10      const char *string2 = "1234567890";
11
12      printf( "%s%s\n%s%s\n\n%s\n%s%u",
13              "string1 = ", string1, "string2 = ", string2,
14              "The length of the initial segment of string1",
15              "containing no characters from string2 = ",
16              strcspn( string1, string2 ) );
17
18      return 0; /* indica che il programma è terminato con successo */
19
20   } /* fine della funzione main */
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

Figura 8.24 Usare strcspn

La funzione `strpbrk` ricerca la prima occorrenza di un carattere qualsiasi della stringa indicata dal suo secondo argomento in quella indicata dal primo. Nel caso che un carattere tra quelli inclusi nella stringa indicata dal secondo argomento sia stato ritrovato, `strpbrk` restituirà un puntatore alla locazione di quel carattere nella stringa indicata dal primo argomento, altrimenti `strpbrk` restituirà un puntatore `NULL`. La Figura 8.25 illustra un programma che individuerà in `string1` la prima occorrenza di uno qualsiasi dei caratteri contenuti in `string2`.

```
 1   /* Fig. 8.25: fig08_25.c
 2      Usare strpbrk */
 3   #include <stdio.h>
 4   #include <string.h>
 5
 6   int main()
 7   {
 8     const char *string1 = "This is a test"; /* inizializza un puntatore
                                                    a carattere */
 9      const char *string2 = "beware"; /* inizializza un puntatore
                                           a carattere */
10
11      printf( "%s\"%s\"\n'%c'%s\n\"%s\"\n",
12              "Of the characters in ", string2,
13              *strpbrk( string1, string2 ),
14              " appears earliest in ", string1 );
15
16      return 0; /* indica che il programma è terminato con successo */
17
18   } /* fine della funzione main */
```

Figura 8.25 Usare strpbrk (continua)

```
Of the characters in "beware"
'a' appears earliest in
"This is a test"
```

Figura 8.25 Usare strpbrk

La funzione `strrchr` ricerca in una stringa l'ultima occorrenza del carattere specificato. Nel caso che questo sia stato ritrovato, `strrchr` restituirà un puntatore alla locazione di quel carattere nella stringa, altrimenti `strrchr` restituirà un puntatore `NULL`. La Figura 8.26 illustra un programma che ricercherà l'ultima occorrenza del carattere `'z'` nella stringa `"A zoo has many animals including zebras"`.

```
1  /* Fig. 8.26: fig08_26.c
2     Usare strrchr */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     /* inizializza un puntatore a carattere */
9     const char *string1 = "A zoo has many animals including zebras";
10
11    int c = 'z'; /* carattere da ricercare */
12
13    printf( "%s\n%s'%c'%s\"%s\"\n",
14           "The remainder of string1 beginning with the",
15           "last occurrence of character ", c,
16           " is: ", strrchr( string1, c ) );
17
18    return 0; /* indica che il programma è terminato con successo */
19
20 } /* fine della funzione main */
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

Figura 8.26 Usare strrchr

La funzione `strspn` (Figura 8.27) determina la lunghezza della parte iniziale della stringa indicata dal suo primo argomento, che contenga soltanto i caratteri inclusi nella stringa indicata dal suo secondo argomento. La funzione restituirà la lunghezza del segmento.

```
1  /* Fig. 8.27: fig08_27.c
2     Usare strspn */
3  #include <stdio.h>
4  #include <string.h>
5
```

Figura 8.27 Usare strspn (continua)

```
 6  int main()
 7  {
 8     /* inizializza due puntatori a carattere */
 9     const char *string1 = "The value is 3.14159";
10     const char *string2 = "aehi lsTuv ";
11
12     printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
13            "string1 = ", string1, "string2 = ", string2,
14            "The length of the initial segment of string1",
15            "containing only characters from string2 = ",
16            strspn( string1, string2 ) );
17
18     return 0; /* indica che il programma è terminato con successo */
19
20  } /* fine della funzione main */
```

```
string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

Figura 8.27 Usare strspn

La funzione `strstr` ricerca nella stringa indicata dal suo primo argomento la prima occorrenza della stringa indicata dal suo secondo argomento. Nel caso che la seconda stringa sia stata ritrovata all'interno della prima, sarà restituito un puntatore alla locazione della seconda stringa all'interno della prima. Il programma della Figura 8.28 utilizzerà `strstr` per ricercare la stringa `"def"` in `"abcdefabcdef"`.

```
 1  /* Fig. 8.28: fig08_28.c
 2     Usare strstr */
 3  #include <stdio.h>
 4  #include <string.h>
 5
 6  int main()
 7  {
 8    const char *string1 = "abcdefabcdef"; /* stringa in cui effettuare
                                               la ricerca */
 9     const char *string2 = "def"; /* stringa da ricercare */
10
11     printf( "%s%s\n%s%s\n\n%s\n%s%s\n",
12            "string1 = ", string1, "string2 = ", string2,
13            "The remainder of string1 beginning with the",
14            "first occurrence of string2 is: ",
15            strstr( string1, string2 ) );
16
17     return 0; /* indica che il programma è terminato con successo */
```

Figura 8.28 Usare strstr (continua)

```
18
19   } /* fine della funzione main */
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Figura 8.28 Usare strstr

La funzione `strtok` è utilizzata per suddividere una stringa in una serie di *token*. Un token è una sequenza di caratteri individuata da *caratteri di delimitazione* (di solito, spazi o segni di punteggiatura). Per esempio, in una riga di testo ogni parola potrebbe essere considerata un token, mentre gli spazi che separano le parole potrebbero essere considerati i caratteri di delimitazione.

Per suddividere una stringa in token (presupponendo che ne contenga più di uno), sarà necessaria una serie di invocazioni di `strtok`. La prima invocazione di `strtok` conterrà due argomenti: la stringa che dovrà essere suddivisa in token e quella che conterrà i caratteri di delimitazione dei token. Nel programma della Figura 8.29, l'istruzione

```
tokenPtr = strtok( string, " " ); /* inizia a suddividere in token
                                     la frase */
```

assegnerà a `tokenPtr` un puntatore al primo token individuato in `string`. Il secondo argomento di `strtok`, `" "`, indica che i token in `string` saranno delimitati da spazi. La funzione `strtok` ricercherà in `string` il primo carattere che non sia quello di delimitazione (lo spazio). Questo carattere corrisponderà all'inizio del primo token. In seguito, la funzione ricercherà nella stringa il carattere di delimitazione susseguente e lo sostituirà con quello nullo (`'\0'`) per delimitare il token corrente. A questo punto, la funzione `strtok` salverà un puntatore al carattere successivo al token individuato in `string` e restituirà un puntatore a quello corrente.

Le invocazioni successive di `strtok` continueranno a suddividere `string`. Queste chiamate dovranno contenere `NULL` come primo argomento. L'argomento `NULL` indica che l'invocazione di `strtok` dovrà continuare la suddivisione in token, cominciando dalla posizione all'interno di `string` salvata durante l'ultima invocazione di `strtok`. Qualora non ci siano altri token, `strtok` restituirà un puntatore `NULL`. Il programma della Figura 8.29 utilizzerà `strtok` per suddividere in token la stringa `"This is a sentence with 7 tokens"`. Ogni token sarà visualizzato separatamente. Osservate che la funzione `strtok` modifica la stringa ricevuta in input; di conseguenza, qualora questa debba essere ancora utilizzata all'interno del programma dopo l'invocazione di `strtok`, sarà meglio prepararne una copia.

```
1   /* Fig. 8.29: fig08_29.c
2      Usare strtok */
3   #include <stdio.h>
4   #include <string.h>
5
6   int main()
```

Figura 8.29 Usare strtok (continua)

```
 7  {
 8     /* inizializza il vettore string */
 9     const char string[] = "This is a sentence with 7 tokens";
10     char *tokenPtr; /* crea un puntatore a carattere */
11
12     printf( "%s\n%s\n\n%s\n",
13            "The string to be tokenized is:", string,
14            "The tokens are:" );
15
16     tokenPtr = strtok( string, " " ); /* inizia a suddividere
                                            in token la frase */
17
18     /* continua a suddividere in token la frase finché tokenPtr
          non assume il valore NULL */
19     while ( tokenPtr != NULL ) {
20        printf( "%s\n", tokenPtr );
21        tokenPtr = strtok( NULL, " " );
22     } /* fine del comando while */
23
24     return 0; /* indica che il programma è terminato con successo */
25
26  } /* fine della funzione main */
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

Figura 8.29 Usare strtok

8.9 Le funzioni per la manipolazione della memoria incluse nella libreria per la gestione delle stringhe

Le funzioni incluse nella libreria per la gestione delle stringhe che presenteremo in questa sezione effettuano ricerche, manipolazioni e confronti di blocchi di memoria. Queste funzioni trattano i blocchi di memoria come stringhe di caratteri e sono in grado di manipolare qualsiasi blocco di dati. La Figura 8.30 riassume le funzioni per la manipolazione della memoria che sono incluse nella libreria per la gestione delle stringhe. Nella trattazione delle suddette funzioni useremo il termine "oggetto" per far riferimento a un blocco di dati.

I parametri di tipo puntatore utilizzati per queste funzioni saranno dichiarati di tipo `void *`. Nel Capitolo 7, abbiamo visto che un puntatore a un tipo di dato qualsiasi può

essere assegnato direttamente a un puntatore di tipo `void *`, e che questo può essere assegnato direttamente a un puntatore che faccia riferimento a un qualsiasi tipo di dato. È proprio per questo motivo che le suddette funzioni potranno ricevere dei puntatori a un qualsiasi tipo di dato. Dato che non è possibile risolvere il riferimento di un puntatore `void *`, ogni funzione dovrà ricevere un argomento di dimensione che specifichi il numero di caratteri (byte) da elaborare. Per semplicità, gli esempi di questa sezione manipoleranno solo vettori di caratteri (blocchi di caratteri).

Prototipo di funzione	Descrizione della funzione
`void *memcpy( void *s1, const void *s2, size_t n )`	Copia `n` caratteri dall'oggetto puntato da `s2` in quello puntato da `s1`. Restituisce un puntatore all'oggetto risultante.
`void *memmove( void *s1, const void *s2, size_t n )`	Copia `n` caratteri dall'oggetto puntato da `s2` in quello puntato da `s1`. La copia sarà eseguita come se i caratteri dell'oggetto puntato da `s2` fossero ricopiati prima in un vettore temporaneo e, quindi, da questo nell'oggetto puntato da `s1`. Restituisce un puntatore all'oggetto risultante.
`int memcmp( const void *s1, const void *s2, size_t n )`	Confronta i primi `n` caratteri degli oggetti puntati da `s1` e da `s2`. La funzione restituisce un valore uguale, minore o maggiore di `0` qualora `s1` sia rispettivamente uguale, minore o maggiore di `s2`.
`void *memchr( const void *s, int c, size_t n )`	Individua la prima occorrenza di `c` (convertito in un `unsigned char`) nei primi `n` caratteri dell'oggetto puntato da `s`. Qualora `c` sia stato ritrovato, sarà restituito un puntatore alla locazione di `c` nell'oggetto. In caso contrario, sarà restituito un puntatore `NULL`.
`void *memset( void *s, int c, size_t n )`	Copia `c` (convertito in un `unsigned char`) nei primi `n` caratteri dell'oggetto puntato da `s`. Restituisce un puntatore all'oggetto risultante.

Figura 8.30 Le funzioni per la manipolazione della memoria incluse nella libreria per la gestione delle stringhe

La funzione `memcpy` copia un numero specificato di caratteri dall'oggetto puntato dal suo secondo argomento in quello puntato dal primo. La funzione può ricevere un puntatore a un qualsiasi tipo di oggetto. Il risultato di questa funzione sarà indefinito, qualora i due oggetti si sovrappongano in memoria, ovverosia qualora essi appartengano allo stesso oggetto: in questi casi usate `memmove`. Il programma della Figura 8.31 utilizzerà `memcpy` per ricopiare nel vettore `s1` la stringa contenuta in `s2`.

La funzione memmove, come memcpy, copia un numero specificato di byte dall'oggetto puntato dal suo secondo argomento in quello puntato dal primo. La copia sarà eseguita come se i byte del secondo argomento fossero ricopiati prima in un vettore di caratteri temporaneo e, quindi, da questo nel primo argomento. Ciò consentirà la copia di una porzione di una stringa in un'altra parte della stessa.

```
1  /* Fig. 8.31: fig08_31.c
2     Usare memcpy */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char s1[ 17 ]; /* crea il vettore di caratteri s1 */
9     char s2[]  = "Copy this string"; /* inizializza il vettore
                                          di caratteri s2 */
10
11    memcpy( s1, s2, 17 );
12    printf( "%s\n%s\"%s\"\n",
13           "After s2 is copied into s1 with memcpy,",
14           "s1 contains ", s1 );
15
16    return 0; /* indica che il programma è terminato con successo */
17
18 } /* fine della funzione main */
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

Figura 8.31 Usare memcpy

Errore tipico 8.8

Escludendo memmove, *tutte le altre funzioni per la manipolazione delle stringhe che copiano dei caratteri produrranno dei risultati indefiniti, qualora la copia avvenga tra porzioni della stessa stringa.*

Il programma della Figura 8.32 utilizzerà memmove per ricopiare gli ultimi 10 byte del vettore **x** nei primi 10 byte dello stesso.

```
1  /* Fig. 8.32: fig08_32.c
2     Usare memmove */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
```

Figura 8.32 Usare memmove

```
8       char x[] = "Home Sweet Home"; /* inizializza il vettore
        di caratteri x */
9
10      printf( "%s%s\n", "The string in array x before memmove is: ", x );
11      printf( "%s%s\n", "The string in array x after memmove is:  ",
12              memmove( x, &x[ 5 ], 10 ) );
13
14      return 0; /* indica che il programma è terminato con successo */
15
16   } /* fine della funzione main */
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

Figura 8.32 Usare memmove

La funzione memcmp (Figura 8.33) confronta un numero specificato di caratteri del suo primo argomento con quelli corrispondenti del secondo. La funzione restituisce un valore positivo se il primo argomento è maggiore del secondo, 0 se sono uguali e un valore negativo se il primo argomento è inferiore al secondo.

```
1   /* Fig. 8.33: fig08_33.c
2      Usare memcmp */
3   #include <stdio.h>
4   #include <string.h>
5
6   int main()
7   {
8      char s1[] = "ABCDEFG"; /* inizializza il vettore di caratteri s1 */
9      char s2[] = "ABCDXYZ"; /* inizializza il vettore di caratteri s2 */
10
11     printf( "%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n",
12             "s1 = ", s1, "s2 = ", s2,
13             "memcmp( s1, s2, 4 ) = ", memcmp( s1, s2, 4 ),
14             "memcmp( s1, s2, 7 ) = ", memcmp( s1, s2, 7 ),
15             "memcmp( s2, s1, 7 ) = ", memcmp( s2, s1, 7 ) );
16
17     return 0; /* indica che il programma è terminato con successo */
18
19  } /* fine della funzione main */
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp( s1, s2, 4 ) =  0
memcmp( s1, s2, 7 ) = -1
memcmp( s2, s1, 7 ) =  1
```

Figura 8.33 Usare memcmp

La funzione `memchr` ricerca la prima occorrenza di un byte, rappresentato come un `unsigned char`, in un numero specificato di byte di un oggetto. Nel caso che il byte sia stato ritrovato sarà restituito un puntatore alla sua locazione all'interno dell'oggetto; altrimenti sarà restituito un puntatore `NULL`. Il programma della Figura 8.34 ricercherà il carattere (byte) `'r'` nella stringa `"This is a string"`.

```
1  /* Fig. 8.34: fig08_34.c
2     Usare memchr */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char *s = "This is a string"; /* inizializza un puntatore
                                        a carattere */
9
10    printf( "%s\'%c\'%s\"%s\"\n",
11           "The remainder of s after character ", 'r',
12           " is found is ", memchr( s, 'r', 16 ) );
13
14    return 0; /* indica che il programma è terminato con successo */
15
16 } /* fine della funzione main */
```

```
The remainder of s after character 'r' is found is "ring"
```

Figura 8.34 Usare memchr

La funzione `memset` copia il valore del byte indicato dal suo secondo argomento nel numero specificato di byte dell'oggetto puntato dal suo primo argomento. Il programma della Figura 8.35 utilizzerà `memset` per copiare `'b'` nei primi 7 byte di `string1`.

```
1  /* Fig. 8.35: fig08_35.c
2     Usare memset */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     char string1[ 15 ] = "BBBBBBBBBBBBBB"; /* inizializza string1 */
9
10    printf( "string1 = %s\n", string1 );
11    printf( "string1 after memset = %s\n", memset( string1, 'b', 7 ) );
12
13    return 0; /* indica che il programma è terminato con successo */
14
15 } /* fine della funzione main */
```

Figura 8.35 Usare memset (continua)

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

Figura 8.35 Usare memset

8.10 Le altre funzioni della libreria per la gestione delle stringhe

Le ultime due funzioni della libreria per la gestione delle stringhe sono: `strerror` e `strlen`. La Figura 8.36 riassume le funzioni `strerror` e `strlen`.

Prototipo di funzione	Descrizione della funzione
`char *strerror( int errornum )`	Traduce `errornum` in una stringa di testo dipendente dal sistema. Restituisce un puntatore alla stringa.
`size_t strlen( const char *s )`	Determina la lunghezza della stringa `s`. Restituisce il numero di caratteri che precedono il carattere nullo di terminazione.

Figura 8.36 Le altre funzioni incluse nella libreria per la gestione delle stringhe

La funzione `strerror` riceve il codice di un errore, crea una stringa che contiene il relativo messaggio e ne restituisce il puntatore. Il programma della Figura 8.37 mostra l'utilizzo di `strerror`.

```
1  /* Fig. 8.37: fig08_37.c
2     Usare strerror */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     printf( "%s\n", strerror( 2 ) );
9
10    return 0; /* indica che il programma è terminato con successo */
11
12 } /* fine della funzione main */
```

```
No such file or directory
```

Figura 8.37 Usare strerror

Obiettivo portabilità 8.4

I messaggi generati da `strerror` *dipendono dal sistema.*

La funzione `strlen` riceve un argomento di tipo stringa e restituisce il numero dei caratteri che la compongono, escludendo il `NULL` di terminazione. Il programma della Figura 8.38 mostra l'utilizzo di `strlen`.

```
1  /* Fig. 8.38: fig08_38.c
2     Usare strlen */
3  #include <stdio.h>
4  #include <string.h>
5
6  int main()
7  {
8     /* inizializza 3 puntatori a carattere */
9     const char *string1 = "abcdefghijklmnopqrstuvwxyz";
10    const char *string2 = "four";
11    const char *string3 = "Boston";
12
13    printf( "%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n%s\"%s\"%s%lu\n",
14           "The length of ", string1, " is ",
15           ( unsigned long ) strlen( string1 ),
16           "The length of ", string2, " is ",
17           ( unsigned long ) strlen( string2 ),
18           "The length of ", string3, " is ",
19           ( unsigned long ) strlen( string3 ) );
20
21    return 0; /* indica che il programma è terminato con successo */
22
23 } /* fine della funzione main */
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Figura 8.38 Usare strlen

Esercizi di autovalutazione

8.1 Scrivete una singola istruzione che esegua ognuna delle seguenti attività. Supponete che le variabili `c` (che immagazzina un carattere), `x`, `y` e `z` siano di tipo `int`, `d`, `e` e `f` siano di tipo `double`, `ptr` sia di tipo `char *` e i vettori `s1[ 100 ]` e `s2[ 100 ]` siano di tipo `char`.

a) Convertite il carattere contenuto nella variabile `c` in una lettera maiuscola. Assegnate il risultato alla variabile `c`.
b) Determinate se il valore della variabile `c` è una cifra. Utilizzate l'operatore condizionale come mostrato nelle Figure 8.2, 8.3 e 8.4 per visualizzare `" is a "` o `" is not a "` quando stampate il risultato.
c) Convertite la stringa `"1234567"` in un `long` e visualizzatene il valore.
d) Determinate se il valore della variabile `c` è un carattere di controllo. Utilizzate l'operatore condizionale per visualizzare `" is a "` o `" is not a "` quando stampate il risultato.
e) Leggete dalla tastiera una riga di testo e immagazzinatela nel vettore `s1`. Non utilizzate `scanf`.
f) Visualizzate la riga di testo immagazzinata nel vettore `s1`. Non utilizzate `printf`.

g) Assegnate a `ptr` la locazione dell'ultima occorrenza di `c` in `s1`.
h) Visualizzate il valore della variabile **`c`**. Non utilizzate `printf`.
i) Convertite la stringa `"8.63582"` in un `double` e visualizzatene il valore.
j) Determinate se il valore di `c` è una lettera. Utilizzate l'operatore condizionale per visualizzare `" is a "` o `" is not a "`, quando stampate il risultato.
k) Leggete un carattere dalla tastiera e immagazzinatelo nella variabile `c`.
l) Assegnate a `ptr` la locazione della prima occorrenza di `s2` in `s1`.
m) Determinate se il valore della variabile `c` è un carattere stampabile. Utilizzate l'operatore condizionale per visualizzare `" is a "` o `" is not a "`, quando stampate il risultato.
n) Leggete tre valori `double` dalla stringa `"1.27 10.3 9.432"` nelle variabili `d`, `e` e `f`.
o) Copiate nel vettore `s1` la stringa immagazzinata in `s2`.
p) Assegnate a `ptr` la locazione della prima occorrenza in `s1` di qualsiasi carattere di `s2`.
q) Confrontate la stringa in `s1` con quella in `s2`. Visualizzate il risultato.
r) Assegnate a `ptr` la locazione della prima occorrenza di `c` in `s1`.
s) Utilizzate `sprintf` per stampare nel vettore `s1` i valori delle variabili intere `x`, `y` e `z`. Ogni valore dovrà essere stampato in un campo di lunghezza 7.
t) Accodate alla stringa `s1` 10 caratteri di `s2`.
u) Determinate la lunghezza della stringa contenuta in `s1`. Visualizzate il risultato.
v) Convertite la stringa `"-21"` in un `int` e visualizzatene il valore.
w) Assegnate a `ptr` la locazione del primo token di `s2`. I token di `s2` sono separati da virgole (`,`).

8.2 Mostrate due metodi differenti per inizializzare il vettore di caratteri `vowel` con la stringa di vocali `"AEIOU"`.

8.3 Che cosa sarà visualizzato, se lo sarà, dopo che ognuna delle seguenti istruzioni C sarà stata eseguita? Nel caso che un'istruzione contenga un errore, descrivetelo e indicate come correggerlo. Supponete che siano già state effettuate le seguenti dichiarazioni di variabile:

```
char s1[ 50 ] = "jack", s2[ 50 ] = "jill", s3[ 50 ], *sptr;
```

a) `printf( "%c%s", toupper( s1[ 0 ] ), &s1[ 1 ] );`
b) `printf( "%s", strcpy( s3, s2 ) );`
c) `printf( "%s", strcat( strcat( strcpy( s3, s1 ), " and " ), s2 ) );`
d) `printf( "%u", strlen( s1 ) + strlen( s2 ) );`
e) `printf( "%u", strlen( s3 ) );`

8.4 Trovate l'errore in ognuno dei seguenti segmenti di programma e spiegate come correggerlo:

a)
```
char s[ 10 ];
strncpy( s, "hello", 5 );
printf( "%s\n", s );
```
b)
```
printf( "%s", 'a' );
```
c)
```
char s[ 12 ];
strcpy( s, "Welcome Home" );
```
d)
```
if ( strcmp( string1, string2 ) )
printf( "The strings are equal\n" );
```

Risposte agli esercizi di autovalutazione

8.1 a)
```
c = toupper( c );
```
b)
```
printf( "'%c'%sdigit\n",
    c, isdigit( c ) ? " is a " : " is not a " );
```

```
c) printf( "%ld\n", atol( "1234567" ) );
d) printf( "'%c'%scontrol character\n",
      c, iscntrl( c ) ? " is a " : " is not a " );
e) gets( s1 );
f) puts( s1 );
g) ptr = strrchr( s1, c );
h) putchar( c );
i) printf( "%f\n", atof( "8.63582" ) );
j) printf( "'%c'%sletter\n",
      c, isalpha( c ) ? " is a " : " is not a " );
k) c = getchar();
l) ptr = strstr( s1, s2 );
m) printf( "'%c'%sprinting character\n",
      c, isprint( c ) ? " is a " : " is not a " );
n) sscanf( "1.27 10.3 9.432", "%f%f%f", &d, &e, &f );
o) strcpy( s1, s2 );
p) ptr = strpbrk( s1, s2 );
q) printf( "strcmp( s1, s2 ) = %d\n", strcmp( s1, s2 ) );
r) ptr = strchr( s1, c );
s) sprintf( s1, "%7d%7d%7d", x, y, z );
t) strncat( s1, s2, 10 );
u) printf( "strlen(s1) = %u\n", strlen( s1 ) );
v) printf( "%d\n", atoi( "-21" ) );
w) ptr = strtok( s2, "," );
```

8.2
```
char vowel[] = "AEIOU";
char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

8.3
a) `Jack`
b) `jill`
c) `jack and jill`
d) `8`
e) `13`

8.4
a) Errore: la funzione `strncpy` non inserirà un carattere nullo di terminazione nel vettore `s`, perché il suo terzo argomento è uguale alla lunghezza della stringa `"hello"`.
   Correzione: cambiate il terzo argomento di `strncpy` in `6` o assegnate `'\0'` a `s[5]`.
b) Errore: tentativo di visualizzare una costante di tipo carattere come se fosse una stringa.
   Correzione: usate `%c` per inviare in output il carattere, oppure sostituite `'a'` con `"a"`.
c) Errore: il vettore di caratteri `s` non è sufficientemente grande per immagazzinare il carattere nullo di terminazione.
   Correzione: dichiarate il vettore con un numero maggiore di elementi.
d) Errore: la funzione `strcmp` restituisce 0 qualora le stringhe siano uguali; di conseguenza, la condizione del comando `if` risulterà falsa e la `printf` non sarà eseguita.
   Correzione: nella condizione, confrontate il risultato di `strcmp` con `0`.

Esercizi

8.5 Scrivete un programma che prenda in input dalla tastiera un carattere e lo controlli con ognuna delle funzioni incluse nella libreria per la gestione dei caratteri. Il programma dovrà visualizzare il valore restituito da ogni funzione.

8.6 Scrivete un programma che prenda in input una riga di testo con la funzione `gets` e la immagazzini nel vettore di caratteri `s[ 100 ]`. Inviate in output la riga prima in lettere maiuscole e poi in minuscole.

8.7 Scrivete un programma che prenda in input quattro stringhe che rappresentino degli interi, le converta in interi, sommi i valori ottenuti e visualizzi il loro totale.

8.8 Scrivete un programma che prenda in input quattro stringhe che rappresentino dei valori in virgola mobile, le converta in `double`, sommi i valori ottenuti e visualizzi il loro totale.

8.9 Scrivete un programma che utilizzi la funzione `strcmp` per confrontare due stringhe immesse dall'utente. Il programma dovrà stabilire se la prima stringa è minore, uguale o maggiore della seconda.

8.10 Scrivete un programma che utilizzi la funzione `strncmp` per confrontare due stringhe immesse dall'utente. Il programma dovrà ricevere in input il numero di caratteri da confrontare e dovrà stabilire se la prima stringa è minore, uguale o maggiore della seconda.

8.11 Scrivete un programma che utilizzi la generazione di numeri casuali per creare delle frasi. Il programma dovrà utilizzare quattro vettori di puntatori a `char` chiamati `article`, `noun`, `verb` e `preposition` e dovrà creare una frase selezionando una parola a caso da ognuno dei vettori nel seguente ordine: `article`, `noun`, `verb`, `preposition`, `article` e `noun`. Man mano che selezionate le singole parole queste dovranno essere concatenate a quelle precedenti, in un vettore che sia sufficientemente grande per contenere la frase intera. Le parole dovranno essere separate da spazi. Quando la frase finale sarà inviata in output, questa dovrà incominciare con una lettera maiuscola e terminare con un punto. Il programma dovrà generare 20 frasi.

I vettori dovranno essere riempiti come segue: `article` dovrà contenere gli articoli `"the"`, `"a"`, `"one"`, `"some"` e `"any"`; `noun` dovrà contenere i nomi `"boy"`, `"girl"`, `"dog"`, `"town"` e `"car"`; `verb` dovrà contenere i verbi `"drove"`, `"jumped"`, `"ran"`, `"walked"` e `"skipped"`; `preposition` dovrà contenere le preposizioni `"to"`, `"from"`, `"over"`, `"under"` e `"on"`.

Una volta che il suddetto programma sarà stato scritto e sarà funzionante, modificatelo in modo che scriva una breve storia formata da una serie di queste frasi. (Vi piace l'idea di uno scrittore casuale?).

8.12 *(Limerick)* Un limerick è una poesia umoristica di cinque versi in cui il primo e il secondo verso fanno rima con il quinto, mentre il terzo fa rima con il quarto. Usando tecniche simili a quelle sviluppate nell'Esercizio 8.11, scrivete un programma C che produca una serie di limerick a caso. Raffinare questo programma in modo che generi buoni limerick sarà un compito impegnativo, ma il risultato varrà bene la fatica fatta!

8.13 Scrivete un programma che codifichi delle frasi della lingua inglese in pig Latin (Latino del maiale). Il pig Latin è una forma di linguaggio codificato usato spesso per divertimento. Esistono molte versioni dei metodi utilizzati per formare delle frasi in pig Latin. Per semplicità, utilizzate il seguente algoritmo:

> Per formare una frase in pig Latin formulatene una in lingua inglese e suddividetela in parole con la funzione **`strtok`**. Per tradurre ogni parola inglese nella corrispondente in pig Latin, spostate la prima lettera della parola inglese in coda alla stessa e aggiungete le lettere `"ay"`. In questo modo, la parola `"jump"` diventerà `"umpjay"`, `"the"` si trasformerà in `"hetay"` e `"computer"` diventerà `"omputercay"`. Gli spazi tra le parole rimarranno tali. Supponete quanto segue: la frase in inglese consisterà di parole separate da spazi, non ci saranno segni di punteggiatura e tutte le parole saranno formate da due o più lettere. La funzione `printLatinWord` dovrà visualizzare ogni parola. [*Suggerimento*: ogni volta che una chiamata di `strtok` avrà trovato un token, passate il puntatore ottenuto alla funzione `printLatinWord` e visualizzate la parola in pig Latin.]

8.14 Scrivete un programma che prenda in input un numero telefonico in una stringa del formato `(555) 555-5555`. Il programma dovrà utilizzare la funzione `strtok` per estrarre il token del prefisso, quello delle prime tre e delle ultime quattro cifre del numero telefonico. Le sette cifre del numero telefonico dovranno essere concatenate in una stringa. Il programma dovrà convertire il prefisso in un `int` e la stringa del numero telefonico in un `long`. Dovranno essere visualizzate entrambe le informazioni: prefisso e numero telefonico.

8.15 Scrivete un programma che prenda in input una riga di testo, la suddivida in token con la funzione `strtok` e li invii in output in ordine inverso.

8.16 Scrivete un programma che prenda in input dalla tastiera una riga di testo e una stringa di ricerca. Utilizzando la funzione `strstr`, individuate la prima occorrenza della stringa di ricerca nella riga di testo e assegnate la locazione ottenuta alla variabile `searchPtr` di tipo `char *`. Nel caso che la stringa di ricerca sia stata ritrovata, cominciando da questa, visualizzate la riga di testo. Utilizzate quindi nuovamente `strstr` per individuare nella riga di testo la successiva occorrenza della stringa di ricerca. Nel caso che sia stata trovata una seconda occorrenza della stringa di ricerca, visualizzate la riga di testo cominciando dalla seconda occorrenza. [*Suggerimento*: la seconda invocazione di `strstr` dovrà contenere `searchPtr + 1` come suo primo argomento.]

8.17 Scrivete un programma, basato sull'Esercizio 8.16, che prenda in input diverse righe di testo e una stringa di ricerca e utilizzi la funzione `strstr` per determinare il totale delle occorrenze della stringa nelle righe di testo. Visualizzate il risultato.

8.18 Scrivete un programma che prenda in input diverse righe di testo e un carattere da ricercare e utilizzi la funzione `strchr`, per determinare il totale delle occorrenze del carattere nelle righe di testo.

8.19 Scrivete un programma, basato su quello dell'Esercizio 8.18, che prenda in input diverse righe di testo e utilizzi la funzione `strchr` per determinare il totale delle occorrenze nelle righe di testo di ogni carattere incluso nell'alfabeto. Le lettere maiuscole e quelle minuscole dovranno essere contate insieme. Immagazzinate i totali di ogni lettera in un vettore e, una volta che saranno stati determinati, visualizzate i suddetti valori in un formato tabulare.

8.20 Scrivete un programma che prenda in input diverse righe di testo e utilizzi la funzione `strtok` per contare il numero totale delle parole. Supponete che queste siano separate da spazi o da caratteri newline.

8.21 Utilizzate le funzioni di confronto delle stringhe discusse nella Sezione 8.6 e le tecniche di ordinamento dei vettori sviluppate nel Capitolo 6, per scrivere un programma che disponga in ordine alfabetico un elenco di stringhe. Utilizzate i nomi di 10 o 15 città della vostra zona come dati per il programma.

8.22 La tabella nell'Appendice D mostra le rappresentazioni in codice numerico dei caratteri dell'insieme ASCII. Studiate questa tabella e quindi stabilite se ognuna delle seguenti affermazioni sia *vera* o *falsa*.

a) La lettera "`A`" precede la "`B`".
b) La cifra "`9`" precede lo "`0`".
c) I simboli comunemente utilizzati per l'addizione, la sottrazione, la moltiplicazione e la divisione precedono tutte le cifre.
d) Le cifre precedono le lettere.
e) Nel caso che un programma di ordinamento ordinasse delle stringhe in modo ascendente, allora quel programma sistemerebbe il simbolo della parentesi tonda chiusa prima di quello della parentesi tonda aperta.

8.23 Scrivete un programma che legga una serie di stringhe e visualizzi solo quelle che iniziano con la lettera "`b`".

8.24 Scrivete un programma che legga una serie di stringhe e visualizzi solo quelle che terminano con le lettere "`ed`".

8.25 Scrivete un programma che prenda in input un codice ASCII e visualizzi il carattere corrispondente. Modificate questo programma, in modo che generi tutti i possibili codici di tre cifre compresi nell'intervallo da 000 a 255 e tenti di visualizzare il carattere corrispondente. Che cosa succederà quando questo programma sarà eseguito?

8.26 Usando come guida la tabella dei caratteri ASCII dell'Appendice D, scrivete le vostre versioni delle funzioni per la gestione dei caratteri presentate nella Figura 8.1.

8.27 Scrivete le vostre versioni delle funzioni per la conversione delle stringhe in numeri presentate nella Figura 8.5.

8.28 Scrivete due versioni di ognuna delle funzioni per la copia e la concatenazione delle stringhe presentate nella Figura 8.17. La prima versione dovrà utilizzare gli indici di vettore, mentre la seconda dovrà utilizzare i puntatori e la relativa aritmetica.

8.29 Scrivete le vostre versioni delle funzioni `getchar`, `gets`, `putchar` e `puts` descritte nella Figura 8.12.

8.30 Scrivete due versioni per ognuna delle funzioni per la comparazione delle stringhe presentate nella Figura 8.20. La prima versione dovrà utilizzare gli indici di vettore, mentre la seconda versione dovrà utilizzare i puntatori e la relativa aritmetica.

8.31 Scrivete le vostre versioni delle funzioni di ricerca nelle stringhe presentate nella Figura 8.22.

8.32 Scrivete le vostre versioni delle funzioni per la manipolazione dei blocchi di memoria presentate nella Figura 8.30.

8.33 Scrivete due versioni della funzione `strlen` presentata nella Figura 8.36. La prima versione dovrà utilizzare gli indici di vettore, mentre la seconda dovrà utilizzare i puntatori e la relativa aritmetica.

Sezione speciale: esercizi di manipolazione avanzata delle stringhe

Gli esercizi precedenti sono stati incentrati sul testo e progettati per verificare la vostra comprensione dei concetti fondamentali della manipolazione delle stringhe. Questa sezione includerà una collezione di problemi di livello intermedio e avanzato. Il lettore dovrebbe trovare questi problemi impegnativi, ma anche divertenti. La difficoltà dei problemi varierà considerevolmente. Alcuni richiederanno un'ora o due per la scrittura del programma e per l'implementazione. Altri saranno utili per attività di laboratorio che potrebbero richiedere due o tre settimane per lo studio e per l'implementazione. Altri ancora sono progetti che vi impegneranno per un intero trimestre.

8.34 *(Analisi del testo)* La disponibilità dei computer, con le loro capacità di manipolazione delle stringhe, ha prodotto alcuni approcci piuttosto interessanti per analizzare gli scritti dei grandi autori. Molta attenzione è stata concentrata sul sospetto che William Shakespeare non sia mai esistito. Alcuni studiosi ritengono che ci siano valide dimostrazioni secondo le quali Christopher Marlowe avrebbe scritto in realtà i capolavori attribuiti a Shakespeare. I ricercatori hanno utilizzato i computer per trovare delle somiglianze negli scritti di questi due autori. Questo esercizio esaminerà tre metodi per analizzare i testi con il computer.

a) Scrivete un programma che legga diverse righe di testo e visualizzi una tabella indicante il numero di occorrenze nel testo per ogni lettera dell'alfabeto. Per esempio, la frase

```
To be, or not to be: that is the question:
```

contiene una "a", due "b", nessuna "c" ecc.

b) Scrivete un programma che legga diverse righe di testo e visualizzi una tabella indicante il numero di parole formate da una sola lettera, da due, da tre, ecc che appaiono nel testo. Per esempio, la frase

```
Whether 'tis nobler in the mind to suffer
```

contiene

Lunghezza della parola	Occorrenze
1	0
2	2
3	2
4	2 (inclusa 'tis)
5	0
6	2
7	1

c) Scrivete un programma che legga diverse righe di testo e visualizzi una tabella indicante il numero di occorrenze nel testo di ogni parola diversa. La prima versione del vostro programma dovrà includere le parole nella tabella sistemandole nell'ordine in cui queste compaiono nel testo. In seguito, provate una visualizzazione più interessante (e utile) in cui le parole siano ordinate alfabeticamente. Per esempio, le righe

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contengono la parola "to" tre volte, "be" due volte, "or" una ecc.

8.35 *(Elaborazione dei testi)* L'approfondimento dedicato alla manipolazione delle stringhe in questo libro è da attribuire in gran parte all'emozionante sviluppo dell'elaborazione dei testi negli anni recenti. Una funzione importante per i sistemi di elaborazione testi è la *giustificazione tipografica*: l'allineamento delle parole al margine sinistro e a quello destro della pagina. La giustificazione tipografica genera un aspetto professionale del documento, dando l'impressione che sia stato impostato in tipografia piuttosto che preparato su una macchina per scrivere. Sui computer la giustificazione tipografica può essere realizzata inserendo uno o più caratteri di spazio tra ognuna delle parole presenti in una riga, in modo che quella più a destra sia allineata con il margine destro.

Scrivete un programma che legga diverse righe di testo e le visualizzi con una giustificazione tipografica. Supponete che il testo debba essere stampato su un foglio largo 8 pollici e mezzo (215,9 millimetri) e che, a sinistra e a destra della pagina stampata, debba essere lasciato un margine di un pollice (25,4 millimetri). Supponete anche che il computer stampi 10 caratteri per pollice. Di conseguenza, il vostro programma dovrà stampare 6 pollici e mezzo di testo (165,1 millimetri), ovverosia 65 caratteri per riga.

8.36 *(Visualizzare le date in vari formati)* Nella corrispondenza commerciale le date sono visualizzate di solito in molti formati. Due dei formati più comuni sono:

```
21/07/55 e 21 luglio 1955
```

Scrivete un programma che legga una data nel primo formato e la visualizzi nel secondo.

8.37 *(Protezione degli assegni)* Molto spesso i computer sono impiegati in sistemi per la compilazione degli assegni, come le applicazioni per la gestione degli stipendi e per la contabilità del debito. Circolano molte strane storie secondo le quali sarebbero stati stampati (erroneamente) degli assegni per la paga settimanale con cifre che superino il milione di dollari. Quelle strane cifre sono stampate dai sistemi computerizzati per la compilazione degli assegni solo a causa di errori umani e/o di deterioramenti della macchina. I progettisti dei sistemi, naturalmente, fanno tutto il possibile per implementare nei propri sistemi dei controlli che prevengano l'emissione di assegni sbagliati.

Un altro grave problema è l'alterazione intenzionale dell'importo di un assegno da parte di qualcuno che intenda incassarlo in modo fraudolento. La maggior parte dei sistemi computerizzati per la compilazione degli assegni impiegano una tecnica chiamata *protezione degli assegni*, proprio per prevenire l'alterazione dei loro importi.

Gli assegni progettati per la compilazione da parte dei computer contengono un numero fisso di spazi in cui il computer può stampare l'importo. Supponete che l'assegno di uno stipendio contenga otto spazi bianchi nei quali si suppone che il computer debba stampare l'ammontare di una paga settimanale. Nel caso che l'importo sia sostanzioso, allora saranno riempiti tutti gli otto spazi. Per esempio:

```
1,230.60 (ammontare dell'assegno in dollari)
— — — —
12345678 (numeri di posizione)
```

Invece, nel caso che la cifra sia inferiore ai $1000 allora molti degli spazi sarebbero lasciati in bianco. Per esempio,

```
   99.87
— — — —
12345678
```

contiene tre spazi bianchi. Sarebbe sicuramente più facile alterare l'importo dell'assegno, qualora questo fosse stato stampato con degli spazi bianchi. Per prevenire l'alterazione di un assegno, molti sistemi per la loro compilazione inseriscono degli *asterischi iniziali* in modo da proteggere l'importo, come nell'esempio che segue:

```
***99.87
— — — —
12345678
```

Scrivete un programma che prenda in input l'importo in dollari che dovrà essere stampato su un assegno e lo visualizzi nel formato protetto dagli asterischi iniziali, qualora siano necessari. Supponete che per stampare l'importo siano disponibili nove spazi.

8.38 *(Scrivere in lettere l'importo di un assegno)* Continuando la discussione dell'esempio precedente, ribadiamo quanto sia importante progettare dei sistemi per la compilazione degli assegni che prevengano l'alterazione del loro importo. Un metodo di sicurezza comune richiede che l'importo dell'assegno sia scritto in cifre e "dichiarato" anche in lettere. Qualcuno è in grado di alterare l'importo numerico di un assegno, ma è estremamente difficile cambiare l'importo espresso in lettere.

Molti sistemi computerizzati per la compilazione degli assegni non stampano l'importo in lettere. Probabilmente, la causa principale di questa omissione è che molti dei linguaggi di alto livello utilizzati nelle applicazioni commerciali non contengono delle caratteristiche adeguate per la manipolazione delle stringhe. Un'altra ragione è che la logica per scrivere in lettere gli importi degli assegni è alquanto complicata.

Scrivete un programma che prenda in input l'ammontare numerico di un assegno e scriva il suo equivalente in lettere. Per esempio, l'importo 112.43 dovrebbe essere scritto come

```
ONE HUNDRED TWELVE and 43/100
```

8.39 *(Il codice Morse)* Il più famoso schema di codifica è probabilmente il codice Morse, sviluppato da Samuel Morse nel 1832 per il sistema telegrafico. Il codice Morse assegna una serie di punti e di linee a ogni lettera dell'alfabeto, a ogni numero e a pochi caratteri speciali (come il punto, la virgola, i due punti e il punto e virgola). Nei sistemi acustici, il punto è rappresentato da un suono breve e la linea è rappresentata da uno lungo. Con i sistemi basati su segnali luminosi o sull'utilizzo di bandierine, sono utilizzate altre rappresentazioni di punti e linee.

La separazione tra le parole è indicata da uno spazio o, più semplicemente, dall'assenza di un punto o di una linea. Nei sistemi acustici, uno spazio è indicato da un breve periodo durante il quale non è trasmesso alcun suono. La versione internazionale del codice Morse è mostrata nella Figura 8.39.

Scrivete un programma che legga una frase in lingua italiana e la codifichi in codice Morse. Scrivete anche un programma che legga una frase in codice Morse e la converta nell'equivalente in lingua italiana. Utilizzate uno spazio tra le lettere del codice Morse e tre fra le parole.

Carattere	Codice	Carattere	Codice
A	.–	T	–
B	–...	U	..–
C	–.–.	V	...–
D	–..	W	.– –
E	.	X	–..–
F	..–.	Y	–.– –
G	– –.	Z	– –..
H	....		
I	..	. **Numeri**	
J	.– –	1	.– – – –
K	–.–	2	..– –
L	.–..	3	...– –
M	– –	4	....–
N	–.	5	.....
O	– –	6	–....
P	.– –.	7	– –...
Q	– –.–	8	– –..
R	.–.	9	– – – –.
S	...	0	– – – –

Figura 8.39 Le lettere dell'alfabeto espresse nel codice Morse internazionale.

8.40 *(Programma per la conversione metrica)* Scrivete un programma che assista l'utente con le conversioni metriche. Il vostro programma dovrà consentire all'utente di specificare con delle stringhe i nomi delle unità di misura (ovverosia, centimetri, litri, grammi ecc. per il sistema metrico decimale e

pollici, quarti di gallone, libbre ecc. per il sistema anglosassone) e dovrà rispondere a semplici domande come:

```
"A quanti pollici corrispondono 2 metri?"
"A quanti litri corrispondono 10 quarti di gallone?"
```

Il vostro programma dovrà riconoscere le conversioni non valide. Per esempio, la domanda

```
"A quanti piedi corrispondono 5 chilogrammi?"
```

non ha senso perché i `"piedi"` sono un'unità di misura della lunghezza mentre i `"chilogrammi"` sono un'unità di misura del peso.

8.41 *(Lettere di sollecito)* Molte aziende commerciali spendono una gran quantità di tempo e di denaro per il recupero del credito. Il *sollecito* è appunto l'invio a un debitore di ripetute e insistenti richieste, nel tentativo di incassare un credito.

Spesso i computer sono utilizzati per generare automaticamente le lettere di sollecito, con un grado crescente di severità man mano che il credito invecchia. La teoria alla base di ciò è che un credito diventa più difficile da incassare man mano che invecchia e, di conseguenza, le lettere di sollecito devono diventare più minacciose.

Scrivete un programma che contenga i testi di cinque lettere di sollecito con severità crescente. Il vostro programma dovrà accettare come input:

1. Il nome del debitore.
2. L'indirizzo del debitore.
3. Il conto del debitore.
4. La cifra dovuta.
5. L'età della cifra dovuta (ovverosia, un ritardo di un mese, di due mesi ecc.).

Utilizzate l'età della cifra dovuta per selezionare uno dei cinque testi del messaggio e stampate la lettera di sollecito, inserendo in modo appropriato le altre informazioni fornite dall'utente.

Un progetto impegnativo per la manipolazione delle stringhe

8.42 *(Un generatore di cruciverba)* Molte persone hanno risolto almeno una volta un cruciverba, ma poche hanno tentato di generarne uno. Generare un cruciverba è un problema difficile. Lo proponiamo in questo contesto come un progetto per la manipolazione delle stringhe che richieda un ragionamento e uno sforzo sostanziosi. Il programmatore dovrà risolvere molti problemi anche per ottenere il più semplice programma per la generazione dei cruciverba. Per esempio, in che modo si potrà rappresentare la griglia di un cruciverba in un computer? È preferibile utilizzare una serie di stringhe o delle matrici? Il programmatore avrà bisogno di una fonte di parole (ovverosia, un dizionario computerizzato) cui il programma possa fare riferimento in modo diretto. In quale forma dovrebbero essere immagazzinate queste parole, per facilitare la complessa manipolazione richiesta dal programma? Il lettore molto ambizioso vorrà generare anche la porzione delle "definizioni" per il cruciverba, in cui siano stampati i brevi suggerimenti forniti al solutore per ogni parola "orizzontale" e "verticale". Anche stampare semplicemente una versione vuota del cruciverba non è un problema semplice.

Capitolo 9

La formattazione dell'input/output in C

Obiettivi

- Comprendere i flussi di input e di output.
- Essere in grado di usare tutte le capacità di formattazione della stampa.
- Essere in grado di usare tutte le capacità di formattazione dell'input.
- Essere in grado di stampare utilizzando le dimensioni di campo e le precisioni.
- Essere in grado di usare i flag di formattazione della stringa di controllo del formato della `printf`.
- Essere in grado di mandare in output i letterali e le sequenze di escape.

9.1 Introduzione

Una parte importante della soluzione di ogni problema è la presentazione dei risultati. In questo capitolo discuteremo in modo approfondito le caratteristiche di formattazione di `scanf` e `printf`. Queste funzioni prendono i dati dallo *stream dello standard input* e li inviano allo *stream dello standard output*, rispettivamente. Nel Capitolo 8 sono state discusse altre quattro funzioni che utilizzano l'input e l'output standard: `gets`, `puts`, `getchar` e `putchar`. Includete il file di intestazione `<stdio.h>` nei programmi che richiamano queste funzioni.

Molte delle caratteristiche di `printf` e `scanf` sono già state discusse in precedenza in questo libro. Questo capitolo riassumerà quelle caratteristiche e ne introdurrà molte altre. Il Capitolo 11 discuterà molte altre funzioni incluse nella libreria per l'input/output standard (`stdio`).

9.2 Gli stream

Tutto l'input e l'output è eseguito attraverso gli *stream*, che sono delle sequenze di byte. Nelle operazioni di input i byte fluiscono da un dispositivo (per esempio, una tastiera, un disco, una connessione di rete) verso la memoria primaria. Nelle operazioni di output i byte fluiscono dalla memoria primaria verso un dispositivo (per esempio, uno schermo, una stampante, un disco, una connessione di rete ecc.).

Nel momento in cui comincia l'esecuzione di un programma, a questo saranno connessi automaticamente tre stream. Di solito, lo stream dello standard input è connesso alla tastiera e quello dell'output allo schermo. Spesso i sistemi operativi consentono il *redirezionamento* di

questi stream verso altri dispositivi. Il terzo stream, lo *standard error*, è connesso allo schermo. I messaggi di errore saranno inviati in output sullo stream dello standard error. Gli stream saranno discussi in dettaglio nel Capitolo 11, "L'elaborazione dei file".

9.3 Formattare l'output con printf

Con `printf` può essere ottenuta una precisa formattazione dell'output. Ogni invocazione di `printf` contiene una *stringa di controllo del formato* che descrive appunto il formato dell'output. La stringa di controllo del formato è composta da *indicatori di conversione*, *flag*, *dimensioni di campo*, *precisioni* e *caratteri letterali*. Uniti al segno di percentuale (%), questi formano le *specifiche di conversione*. La funzione `printf` ha le seguenti capacità di formattazione, ognuna delle quali sarà discussa in questo capitolo:

1. *Arrotondamento* dei valori in virgola mobile al numero indicato di cifre decimali.
2. *Allineamento* di una colonna di numeri con i separatori dei decimali che compaiono uno sull'altro.
3. *Allineamento a destra* e *a sinistra* degli output.
4. *Inserimento di caratteri letterali* in posizioni precise della riga inviata in output.
5. Rappresentazione dei numeri in virgola mobile in formato esponenziale.
6. Rappresentazione degli interi senza segno in formato ottale ed esadecimale. Consultate l'Appendice E, I sistemi numerici, per ottenere maggiori informazioni sui valori ottali ed esadecimali.
7. Visualizzazione di tutti i tipi di dato in campi di dimensione e precisione prefissate.

La funzione `printf` ha la forma:

`printf` (*stringa di controllo del formato, altri argomenti*);

La *stringa di controllo del formato* descrive il formato dell'output, mentre gli *altri argomenti* (che sono opzionali) corrispondono alle singole specifiche di conversione inserite nella *stringa di controllo del formato*. Ogni specifica di conversione incomincia con un segno di percentuale e termina con un indicatore di conversione. In una stringa di controllo del formato potranno essere inserite molte specifiche di conversione.

Errore tipico 9.1

Dimenticare di racchiudere tra virgolette la stringa di controllo del formato è un errore di sintassi.

Buona abitudine 9.1

Curate la formattazione dell'output in modo da ottenere una presentazione ordinata per rendere più leggibili gli output del vostro programma e ridurre gli errori dell'utente.

9.4 Visualizzare gli interi

Un intero è un numero, come `776`, `0` o `-52`, che non contiene virgole decimali. I valori interi possono essere visualizzati in molti formati. La Figura 9.1 descrive le specifiche di conversione disponibili per gli interi.

Il programma della Figura 9.2 visualizzerà un intero utilizzando ogni indicatore di conversione disponibile. Osservate che sarà stampato soltanto il segno negativo; quelli positivi sono soppressi. Più tardi in questo capitolo vedremo in che modo sarà possibile forzare la visualizzazione dei segni positivi. Notate anche che `-455` quando sarà letto per mezzo di `%u`, verrà convertito nel valore senza segno `4294966841`.

Indicatore di conversione	Descrizione
`d`	Visualizza un intero decimale con segno.
`i`	Visualizza un intero decimale con segno. [*Nota*: il comportamento degli indicatori `i` e `d` sarà diverso quando saranno utilizzati con `scanf`].
`o`	Visualizza un intero ottale senza segno.
`u`	Visualizza un intero decimale senza segno.
`x` o `X`	Visualizza un intero esadecimale senza segno. `X` induce la visualizzazione delle cifre `0-9` e delle lettere `A-F`, mentre `x` induce la visualizzazione delle cifre `0-9` e delle lettere `a-f`.
`h` o `l` (lettera `l`)	Posto dinanzi a ogni indicatore di conversione per gli interi, per indicare che sarà visualizzato rispettivamente un intero `short` o `long`. Più precisamente le lettere `h` e `l` sono chiamate *modificatori di lunghezza*.

Figura 9.1 Gli indicatori di conversione per gli interi

```
1   /* Fig 9.2: fig09_02.c
2      Usare gli indicatori di conversione per gli interi */
3   #include <stdio.h>
4
5   int main()
6   {
7     printf("%d\n", 455);
8     printf("%i\n", 455);  /* i è come d nella printf */
9     printf("%d\n", +455);
10    printf(„%d\n", -455);
11    printf(„%hd\n", 32000);
12    printf(„%ld\n", 2000000000);
13    printf("%o\n", 455);
14    printf("%u\n", 455);
15    printf("%u\n", -455);
16    printf("%x\n", 455);
17    printf("%X\n", 455);
18
19    return 0; /* indica che il programma è terminato con successo */
20
21  } /* fine della funzione main */
```

Figura 9.2 Usare gli indicatori di conversione per gli interi (continua)

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Figura 9.2 Usare gli indicatori di conversione per gli interi

Errore tipico 9.2

Visualizzare un valore negativo con un indicatore di conversione che si aspetta un valore `unsigned`.

9.5 Visualizzare i numeri in virgola mobile

Un valore in virgola mobile contiene una virgola decimale come `33,5`, `0,0` o `657,983`. I valori in virgola mobile potranno essere visualizzati in molti formati. La Figura 9.3 descrive gli indicatori di conversione disponibili per i numeri in virgola mobile.

Indicatore di conversione	Descrizione
`e` o `E`	Visualizza un valore in virgola mobile nella notazione esponenziale.
`f`	Visualizza un valore in virgola mobile nella notazione in virgola fissa.
`g` o `G`	Visualizza un valore in virgola mobile sia nel formato di `f` che in quello di `e` (o di `E`) in base alla grandezza del valore.
`L`	Posto dinanzi a un qualsiasi indicatore di conversione per i numeri in virgola mobile, per indicare che sarà visualizzato un valore in virgola mobile `long double`.

Figura 9.3 Gli indicatori di conversione per i valori in virgola mobile

Gli indicatori di conversione `e` ed `E` visualizzano i valori in virgola mobile nella *notazione esponenziale*. Nei computer, la notazione esponenziale è l'equivalente della *notazione scientifica* utilizzata in matematica. Per esempio, il valore 150,4582 nella notazione scientifica è rappresentato come

```
1,504582 × 102
```

mentre in quella esponenziale è rappresentato dal computer come

```
1,504582E+02
```

Questa notazione indica che `1,504582` è moltiplicato per `10` elevato alla seconda potenza `(E+02)`. La `E` sta appunto per "esponente".

I valori visualizzati con gli indicatori di conversione `e`, `E` e `f` saranno visualizzati per default con 6 cifre di precisione a destra della virgola decimale (per esempio, `1,04592`); le altre precisioni potranno essere specificate in modo esplicito. L'indicatore di conversione `f` visualizza sempre almeno una cifra a sinistra della virgola decimale. Gli indicatori di conversione `e` ed `E` visualizzano rispettivamente una `e` minuscola e una `E` maiuscola davanti all'esponente, e stamperanno sempre e solo una cifra a sinistra della virgola decimale.

L'indicatore di conversione `g` (`G`) visualizza i valori nei formati `e` (`E`) o `f`, ma eliminando gli zeri finali nella parte decimale (in altri termini, `1,234000` sarebbe visualizzato come `1,234`). I valori saranno visualizzati nel formato `e` (`E`) qualora, dopo la loro conversione in notazione esponenziale, l'esponente sia inferiore a `-4` oppure sia maggiore o uguale alla precisione specificata (per default 6 cifre significative per `g` e `G`). In caso contrario, per visualizzare i valori sarà utilizzato l'indicatore di conversione `f`. Nella parte frazionale di un valore inviato in output con `g` o `G` gli zeri finali non saranno visualizzati. Perché sia visualizzata la virgola dei decimali sarà necessaria almeno una cifra decimale. Con l'indicatore di conversione `g`, i valori `0,0000875`, `8750000,0`, `8,75`, `87,50` e `875` sarebbero visualizzati come `8,75e-05`, `8,75e+06`, `8,75`, `87,5` e `875`. Il valore `0,0000875` utilizzerà la notazione `e` perché, quando sarà stato convertito in quella esponenziale, il suo esponente (`-5`) sarà minore di `-4`. Il valore `8750000,0` utilizzerà la notazione `e` perché il suo esponente (`6`) sarà uguale alla precisione di default.

Per gli indicatori di conversione `g` e `G`, la precisione indica il numero massimo di cifre significative che saranno visualizzate, inclusa quella a sinistra della virgola decimale. Utilizzando la specifica di conversione `%g`, il valore `1234567,0` sarà visualizzato come `1,23457e+06` (ricordate che tutti gli indicatori di conversione per i numeri in virgola mobile hanno per default una precisione di 6 cifre). Osservate che nel risultato ci saranno appunto 6 cifre significative. Nel caso che il valore sia visualizzato in notazione esponenziale, la differenza tra `g` e `G` sarà la stessa che intercorre tra `e` ed `E`: la `g` minuscola visualizzerà una `e` minuscola, mentre la `G` maiuscola stamperà una `E` maiuscola.

Il programma della Figura 9.4 mostrerà l'utilizzo delle tre specifiche di conversione disponibili per i valori in virgola mobile. Osservate che le specifiche di conversione `%E`, `%e` e `%g` arrotonderanno il valore inviato in output, mentre la specifica di conversione `%f` non lo farà.

```
1   /* Fig 9.4: fig09_04.c */
2   /* Visualizzare i numeri in virgola mobile con gli indicatori
3      di conversione per i valori in virgola mobile */
4
5   #include <stdio.h>
6
7   int main()
8   {
9     printf("%e\n", 1234567.89);
10    printf("%e\n", +1234567.89);
```

Figura 9.4 Usare gli indicatori di conversione per i valori in virgola mobile (continua)

```
11     printf("%e\n", -1234567.89);
12     printf("%E\n", 1234567.89);
13     printf("%f\n", 1234567.89);
14     printf("%g\n", 1234567.89);
15     printf("%G\n", 1234567.89);
16
17     return 0; /* indica che il programma è terminato con successo */
18
19  } /* fine della funzione main */
```

```
1.234568e+06
1.234568e+06
-1.234568e+06
1.234568E+06
1234567.890000
1.23457e+06
1.23457E+06
```

Figura 9.4 Usare gli indicatori di conversione per i valori in virgola mobile

Collaudo e messa a punto 9.1

Quando inviate i dati in output, assicuratevi di informare l'utente delle situazioni in cui i dati potrebbero non essere precisi a causa della formattazione (per esempio, per gli errori di arrotondamento dovuti alla precisione specificata).

9.6 Visualizzare le stringhe e i caratteri

Gli indicatori di conversione c e s sono utilizzati rispettivamente per visualizzare i singoli caratteri e le stringhe. L'indicatore di conversione c richiede un argomento di tipo char, mentre s richiede un puntatore a char. L'indicatore di conversione s provocherà la visualizzazione dei caratteri finché non sarà stato incontrato il carattere nullo ('\0') di terminazione della stringa. Il programma mostrato nella Figura 9.5 visualizzerà alcuni caratteri e stringhe utilizzando gli indicatori di conversione c e s.

```
1  /* Fig 9.5: fig09_05.c */
2  /*  Visualizzare stringhe e caratteri */
3  #include <stdio.h>
4
5  int main()
6  {
7     char character = 'A'; /* inizializza la variabile di tipo char */
8     char string[] = "This is a string"; /* inizializza
                                  il vettore di elementi di tipo char */
9     const char *stringPtr = "This is also a string";
                                          */ puntatore a char */
10
11    printf("%c\n", character);
```

Figura 9.5 Utilizzare gli indicatori di conversione per i caratteri e le stringhe (continua)

```
12     printf("%s\n", "This is a string");
13     printf("%s\n", string);
14     printf("%s\n", stringPtr);
15
16     return 0; /* indica che il programma è terminato con successo */
17
18   } /* fine della funzione main */
```

```
A
This is a string
This is also a string
```

Figura 9.5 Utilizzare gli indicatori di conversione per i caratteri e le stringhe

Errore tipico 9.3

Utilizzare `%c` *per visualizzare una stringa è un errore. La specifica di conversione* `%c` *si attende un argomento di tipo* `char`. *Una stringa invece è un puntatore a* `char` *(ossia un* `char *`*).*

Errore tipico 9.4

Utilizzare `%s` *per visualizzare un argomento di tipo* `char`, *su alcuni sistemi, provocherà un errore fatale detto violazione di accesso. La specifica di conversione* `%s` *si aspetta un argomento di tipo puntatore a* `char`.

Errore tipico 9.5

È un errore di sintassi usare degli apici singoli per delimitare le stringhe di caratteri. Queste, infatti, devono essere racchiuse da virgolette.

Errore tipico 9.6

Utilizzare le virgolette intorno alle costanti di carattere creerà in realtà una stringa formata da due caratteri, il secondo dei quali sarà il carattere nullo di terminazione. Una costante di carattere è invece un singolo carattere racchiuso tra apici singoli.

9.7 Gli altri indicatori di conversione

Gli ultimi tre indicatori di conversione rimasti sono: `p`, `n` e `%` (Figura 9.6).

Indicatore di conversione	Descrizione
`p`	Visualizza il valore di un puntatore in un formato definito dall'implementazione.
`n`	Immagazzina il numero di caratteri già inviato in output dall'istruzione `printf` corrente. Come argomento corrispondente dovrà essere fornito un puntatore a un intero. L'indicatore di conversione non visualizzerà niente.
`%`	Visualizza il simbolo di percentuale.

Figura 9.6 Gli altri indicatori di conversione

Obiettivo portabilità 9.1

L'indicatore di conversione p *visualizza l'indirizzo di un puntatore in un formato definito dall'implementazione (su molti sistemi, è utilizzata la notazione esadecimale piuttosto che quella decimale).*

L'indicatore di conversione n immagazzina il numero di caratteri già inviati in output dall'istruzione printf corrente: infatti, l'argomento corrispondente è un puntatore alla variabile intera nella quale sarà immagazzinato il valore. La specifica di conversione %n non visualizza niente. L'indicatore di conversione % visualizza un segno di percentuale.

In Figura 9.7, %p visualizzerà il valore di ptr e l'indirizzo di x; questi dati saranno identici poiché l'indirizzo di x sarà stato assegnato a ptr. In seguito, %n immagazzinerà nella variabile intera y il numero dei caratteri che saranno stati inviati in output dalla terza istruzione printf (riga 15) e il valore di y sarà visualizzato. L'ultima istruzione printf (riga 21) utilizzerà %% per visualizzare il carattere % all'interno di una stringa. Osservate che ogni chiamata di printf restituisce un valore: il numero dei caratteri inviati in output o un valore negativo, qualora si sia verificato un errore nell'output.

Errore tipico 9.7

Tentare di visualizzare il simbolo di percentuale usando % invece di %% nella stringa di controllo del formato. Nel momento in cui in una stringa di controllo del formato appare un %, questo deve essere seguito da un indicatore di conversione.

```
1   /* Fig 9.7: fig09_07.c */
2   /* Usare gli indicatori di conversione p, n, e %  */
3   #include <stdio.h>
4
5   int main()
6   {
7     int *ptr;        /* dichiara un puntatore al tipo int */
8     int x = 12345;         /* inizializza la variabile x di tipo int
9     int y;                 /* dichiara la variabile y di tipo int */
10
11    ptr = &x;        /* assegna l'indirizzo di x a ptr */
12    printf("The value of ptr is %p\n", ptr);
13    printf("The address of x is %p\n\n", &x);
14
15    printf("Total characters printed on this line is:%n", &y);
16    printf(" %d\n\n", y);
17
18    y = printf("This line has 28 characters\n");
19    printf("%d characters were printed\n\n", y);
20
21    printf("Printing a %% in a format control string\n");
22
23    return 0; /* indica che il programma è terminato con successo */
24
25  } /* fine della funzione main */
```

Figura 9.7 Utilizzare gli indicatori di conversione p, n e % (continua)

```
The value of ptr is 0012FF78
The address of x is 0012FF78

Total characters printed on this line is: 38

This line has 28 characters
28 characters were printed

Printing a % in a format control string
```

Figura 9.7 Utilizzare gli indicatori di conversione p, n e %

9.8 Visualizzare con le dimensioni di campo e le precisioni

L'esatta misura del campo in cui saranno visualizzati i dati è specificata dalla cosiddetta *dimensione di campo*. Nel caso che la dimensione del campo sia maggiore del dato da visualizzare questo, di norma, sarà allineato a destra all'interno di quel campo. Normalmente l'intero che rappresenta la dimensione del campo è inserito tra il segno di percentuale (%) e l'indicatore di conversione (ad esempio, %4d). Il programma della Figura 9.8 visualizzerà due gruppi di cinque numeri, allineando a destra quelli che contengono meno cifre della dimensione del campo. Osservate che, per visualizzare dei valori con una dimensione maggiore di quella del campo, questa sarà incrementata, e che il segno dei valori negativi utilizzerà una posizione all'interno della stessa. Le dimensioni dei campi potranno essere utilizzate con tutti gli indicatori di conversione.

Errore tipico 9.8

Non fornire una dimensione di campo sufficientemente grande per gestire il valore da visualizzare potrebbe spostare gli altri dati da visualizzare e produrre degli output disordinati. Studiate i vostri dati!

```
1   /* Fig 9.8: fig09_08.c */
2   /* Visualizzare gli interi allineati a destra */
3   #include <stdio.h>
4
5   int main()
6   {
7     printf("%4d\n", 1);
8     printf("%4d\n", 12);
9     printf("%4d\n", 123);
10    printf("%4d\n", 1234);
11    printf("%4d\n\n", 12345);
12
13    printf("%4d\n", -1);
14    printf("%4d\n", -12);
15    printf("%4d\n", -123);
```

Figura 9.8 Allineare a destra in un campo gli interi (continua)

```
16      printf("%4d\n", -1234);
17      printf("%4d\n", -12345);
18
19      return 0; /* indica che il programma è terminato con successo */
20
21   } /* fine della funzione main */
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Figura 9.8 Allineare a destra in un campo gli interi

La funzione `printf` fornisce anche la possibilità di specificare la *precisione* con cui il dato dovrà essere visualizzato. La precisione ha un significato diverso per i vari tipi di dato. Utilizzata insieme agli indicatori di conversione per gli interi, la precisione indica il numero minimo di cifre da visualizzare. Nel caso che il valore visualizzato contenga meno cifre di quelle indicate dalla precisione, davanti al suddetto valore saranno inseriti degli zeri finché il numero totale delle cifre non risulti equivalente a quello della precisione. La precisione predefinita per gli interi è 1. Utilizzata con gli indicatori di conversione `e`, `E` e `f` per i valori in virgola mobile, la precisione indica il numero di cifre che dovranno comparire dopo la virgola dei decimali. Utilizzata con gli indicatori di conversione `g` e `G`, la precisione indica il numero massimo di cifre significative da visualizzare. Utilizzata con l'indicatore di conversione `s`, la precisione indica il numero massimo di caratteri della stringa che dovranno essere scritti. Per utilizzare la precisione, inserirete un punto (`.`) seguito dall'intero che rappresenta la precisione, tra il segno di percentuale e l'indicatore di conversione. In Figura 9.9 verrà mostrato l'utilizzo della precisione nelle stringhe di controllo del formato. Osservate che un valore in virgola mobile sarà arrotondato, qualora sia visualizzato con una precisione inferiore al numero originale delle sue cifre decimali.

```
1  /* Fig 9.9: fig09_09.c */
2  /* Usare la precisione mentre si visualizzano gli interi,
3      i numeri in virgola mobile e le stringhe */
4  #include <stdio.h>
5
6  int main()
7  {
8     int i = 873;          /* inizializza la variabile I di tipo int */
```

Figura 9.9 Utilizzare le precisioni per visualizzare informazioni di tipo differente (continua)

```
 9      float f = 123.94536; /* inizializza la variabile f
                                di tipo double */
10      char s[] = "Happy Birthday";      /* inizializza il vettore
                                             di caratteri s */
11
12      printf("Using precision for integers\n");
13      printf("\t%.4d\n\t%.9d\n\n", i, i);
14
15      printf("Using precision for floating-point numbers\n");
16      printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);
17
18      printf(„Using precision for strings\n");
19      printf("\t%.11s\n", s);
20
21      return 0; /* indica che il programma è terminato con successo */
22
23   } /* fine della funzione main */
```

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```

Figura 9.9 Utilizzare le precisioni per visualizzare informazioni di tipo differente

La dimensione del campo e la precisione potranno essere combinate inserendo, tra il segno di percentuale e l'indicatore di conversione, la dimensione del campo seguita dal punto e dalla precisione, come nell'istruzione

```
printf("%9.3f", 123.456789);
```

che visualizzerà il valore `123,457` con tre cifre dopo la virgola dei decimali e allineato a destra in un campo di nove posizioni.

Sarà anche possibile specificare la dimensione di campo e la precisione, utilizzando delle espressioni intere inserite nella lista degli argomenti che segue la stringa di controllo del formato. Per utilizzare questa caratteristica, inserirete un * (asterisco) al posto della dimensione di campo o della precisione (o di entrambe). In sostituzione dell'asterisco sarà valutato e utilizzato l'argomento di tipo `int` corrispondente. Un valore per la dimensione del campo potrà essere positivo o negativo (in quest'ultimo caso ciò provocherà l'allineamento a sinistra dell'output, come descritto nella sezione successiva). L'istruzione

```
printf("%*.*f", 7, 2, 98.736);
```

utilizzerà `7` per la dimensione del campo, `2` per la precisione e visualizzerà il valore `98,74` allineato a destra.

9.9 Utilizzare i flag nella stringa di controllo del formato della printf

La funzione `printf` fornisce anche dei *flag* (segnalini o bandierine) per integrare le proprie capacità di formattazione. Ci sono cinque flag da poter utilizzare nelle stringhe di controllo del formato (Figura 9.10).

Flag	Descrizione
- (segno meno)	Allinea a sinistra l'output all'interno del campo specificato.
+ (segno più)	Visualizza il segno davanti ai valori positivi e a quelli negativi.
spazio	Visualizza uno spazio davanti a un valore positivo che non sia stato visualizzato con il flag +.
#	Visualizza uno `0` davanti al valore, qualora sia utilizzato insieme all'indicatore di conversione in ottale `o`.
	Visualizza `0x` o `0X` davanti al valore, qualora sia utilizzato insieme agli indicatori di conversione in esadecimale `x` o `X`.
	Forza la visualizzazione di una virgola decimale per i numeri in virgola mobile che siano stati visualizzati con `e`, `E`, `f`, `g` o `G` e che non contengano una parte frazionaria. (Normalmente, la virgola decimale sarà visualizzata soltanto qualora sia seguita da una cifra.) Per gli indicatori `g` e `G`, gli zeri in coda al valore non saranno eliminati.
`0` (zero)	Riempie un campo inserendo degli zeri davanti al valore visualizzato.

Figura 9.10 I flag per la stringa di controllo del formato

Per utilizzare un flag in una stringa di controllo del formato, dovrete inserirlo immediatamente a destra del segno di percentuale. All'interno di una specifica di conversione potranno essere combinati molti flag.

Il programma della Figura 9.11 mostrerà l'utilizzo dell'allineamento a destra e a sinistra di una stringa, di un intero, di un carattere e di un numero in virgola mobile.

Il programma della Figura 9.12 visualizzerà un numero positivo e uno negativo, con e senza il *flag* +. Notate che il segno negativo sarà visualizzato in entrambi i casi, mentre quello positivo sarà stampato soltanto quando sarà stato utilizzato il flag +.

Il programma della Figura 9.13 inserirà uno spazio davanti al numero positivo, usando l'*apposito flag*, che sarà molto utile per allineare dei numeri positivi e negativi che abbiano la stessa quantità di cifre. Si osservi che il valore `-547` non verrà preceduto da uno spazio in output a causa del suo segno meno.

```
1  /* Fig 9.11: fig09_11.c */
2  /* Allineare dei valori a destra e a sinistra */
3  #include <stdio.h>
4
5  int main()
```

Figura 9.11 Allineare a sinistra le stringhe in un campo (continua)

```
 6   {
 7     printf("%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23);
 8     printf("%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23);
 9
10      return 0; /* indica che il programma è terminato con successo */
11
12   } /* fine della funzione main */
```

```
     hello         7         a  1.230000

hello     7         a         1.230000
```

Figura 9.11 Allineare a sinistra le stringhe in un campo

```
 1   /* Fig 9.12: fig09_12.c */
 2   /* Visualizzare dei numeri con e senza il flag + */
 3   #include <stdio.h>
 4
 5   int main()
 6   {
 7     printf("%d\n%d\n", 786, -786);
 8     printf("%+d\n%+d\n", 786, -786);
 9
10     return 0; /* indica che il programma è terminato con successo */
11
12   } /* fine della funzione main */
```

```
786
-786
+786
-786
```

Figura 9.12 Visualizzare dei numeri positivi e negativi con e senza il flag +

```
 1   /* Fig 9.13: fig09_13.c */
 2   /* Visualizzare uno spazio davanti a valori con segno
 3       che non siano preceduti da + o - */
 4   #include <stdio.h>
 5
 6   int main()
 7   {
 8     printf("% d\n% d\n", 547, -547);
 9
10     return 0; /* indica che il programma è terminato con successo */
11
12   } /* fine della funzione main */
```

```
 547
-547
```

Figura 9.13 Usare il flag spazio

Il programma nella Figura 9.14 utilizzerà il *flag* # per inserire uno 0 davanti al valore ottale, 0x e 0X davanti a quelli esadecimali e per forzare la virgola dei decimali in un valore visualizzato con g.

```
1   /* Fig 9.14: fig09_14.c */
2   /* Usare il flag # con gli indicatori di conversione o, x, X
3      e ogni indicatore di conversione per numeri in virgola
       mobile */
4   #include <stdio.h>
5
6   int main()
7   {
8     int c = 1427; /* inizializza c /*
9     double p = 1427.0; /* inizializza p */
10
11    printf("%#o\n", c);
12    printf("%#x\n", c);
13    printf("%#X\n", c);
14    printf("\n%g\n", p);
15    printf("%#g\n", p);
16
17    return 0; /* indica che il programma è terminato con successo */
18
19  } /* fine della funzione main */
```

```
02623
0x593
0X593

1427
1427.00
```

Figura 9.14 Usare il flag #

Il programma della Figura 9.15 combinerà i flag + e 0 (zero) per visualizzare 452 in un campo con 9 posizioni, con un segno + e degli zeri davanti al valore; in seguito visualizzerà nuovamente il 452 utilizzando soltanto il flag 0 e un campo con 9 posizioni.

```
1   /* Fig 9.15: fig09_15.c */
2   /* Usare il flag 0 (zero) serve a completare il campo
       con degli zeri davanti al valore da visualizzare */
3   #include <stdio.h>
4
5   int main()
6   {
7     printf("%+09d\n", 452);
8     printf("%09d", 452);
```

Figura 9.15 Utilizzare il flag 0 (zero) (continua)

```
 9
10     return 0; /* indica che il programma è terminato con successo */
11
12   } /* fine della funzione main */
```

```
+00000452
000000452
```

Figura 9.15 Utilizzare il flag 0 (zero)

9.10 Visualizzare i letterali e le sequenze di escape

Per essere visualizzati da un'istruzione `printf`, la maggior parte dei caratteri letterali dovranno semplicemente essere inclusi nella stringa di controllo. Esistono tuttavia molti caratteri "problematici", come le virgolette (") che delimitano la stessa stringa di controllo del formato. Vari caratteri di controllo, come il newline e la tabulazione, dovranno essere rappresentati da *sequenze di escape*. Queste sono rappresentate da un backslash (\) seguito da un particolare *carattere di escape*. La Figura 9.16 elenca tutte le sequenze di escape e le azioni che provocheranno.

Sequenza di escape	Descrizione
`\'` (apice singolo)	Visualizza il carattere apice singolo (').
`\"` (virgolette)	Visualizza il carattere virgolette (").
`\?` (punto interrogativo)	Visualizza il carattere punto interrogativo (?).
`\\` (backslash)	Visualizza il carattere backslash (\).
`\a` (allarme o cicalino)	Provoca un allarme acustico (cicalino) o visivo.
`\b` (backspace)	Muove il cursore indietro di una posizione sulla riga corrente.
`\f` (pagina nuova o avanzamento del foglio)	Muove il cursore all'inizio della successiva pagina logica.
`\n` (linea nuova)	Muove il cursore all'inizio della riga successiva.
`\r` (ritorno del carrello)	Muove il cursore all'inizio della riga corrente.
`\t` (spaziatura orizzontale)	Muove il cursore sulla successiva posizione di tabulazione orizzontale.
`\v` (spaziatura verticale)	Muove il cursore sulla successiva posizione di tabulazione verticale.

Figura 9.16 Sequenze di escape

Errore tipico 9.9

Tentare di visualizzare come dato letterale in un'istruzione `printf` *un apice singolo, delle virgolette, un punto interrogativo o il carattere backslash, senza farli precedere dal backslash per formare una sequenza di escape appropriata è un errore.*

9.11 Formattare l'input con scanf

Con `scanf` potrà essere ottenuta una formattazione precisa dell'input. Ogni istruzione `scanf` contiene una stringa di controllo che descrive il formato dei dati da prendere in input. La stringa di controllo del formato consiste di specifiche di conversione e di caratteri letterali. La funzione `scanf` ha le seguenti capacità di formattazione:

1. Prendere in input tutti i tipi di dato.
2. Prendere da uno stream di input dei caratteri specifici.
3. Ignorare dei caratteri specifici dallo stream di input.

La funzione `scanf` è scritta nel seguente formato:

`scanf(`*stringa di controllo del formato, altri argomenti*`);`

La *stringa di controllo del formato* descrive appunto i formati per l'input, mentre gli *altri argomenti* sono puntatori alle variabili nelle quali saranno immagazzinati i dati ricevuti in input.

Buona abitudine 9.2

Richiedete all'utente uno o pochi dati per volta. Evitate di richiedere l'immissione di molti dati a fronte di una singola richiesta.

La Figura 9.17 riassume gli indicatori di conversione utilizzati per prendere in input tutti i tipi di dato. La parte rimanente di questa sezione proporrà alcuni programmi che illustreranno la lettura dei dati eseguita con vari indicatori di conversione per la `scanf`.

Indicatori di conversione	Descrizione
Interi	
`d`	Legge un intero decimale con o senza segno. L'argomento corrispondente è un puntatore a un intero.
`i`	Legge un intero decimale, ottale o esadecimale con o senza segno. L'argomento corrispondente è un puntatore a un intero.
`o`	Legge un intero ottale. L'argomento corrispondente è un puntatore a un intero senza segno.
`u`	Legge un intero decimale senza segno. L'argomento corrispondente è un puntatore a un intero senza segno.
`x` o `X`	Legge un intero esadecimale. L'argomento corrispondente è un puntatore a un intero senza segno.
`h` o `l`	Posto dinanzi a uno qualsiasi degli indicatori di conversione per gli interi, indica che sarà preso in input un intero `short` o `long`.
Numeri in virgola mobile	
`e`, `E`, `f`, `g` o `G`	Legge un valore in virgola mobile. L'argomento corrispondente è un puntatore a una variabile in virgola mobile.

Figura 9.17 Gli indicatori di conversione per scanf (continua)

Indicatori di conversione	Descrizione
`l` o `L`	Posto dinanzi a uno qualsiasi degli indicatori di conversione per numeri in virgola mobile, indica che sarà preso in input un valore `double` o `long double`. L'argomento corrispondente è un puntatore a una variabile di tipo `double` o `long double`.
Caratteri e stringhe	
`c`	Legge un carattere. L'argomento corrispondente è un puntatore a `char`; il carattere nullo (`'\0'`) non sarà aggiunto.
`s`	Legge una stringa. L'argomento corrispondente è un puntatore a un vettore di tipo `char`, che sia sufficientemente grande per contenere la stringa e un carattere nullo (`'\0'`) di terminazione che viene aggiunto automaticamente.
Gruppo di scansione	
[*caratteri di scansione*]	Scandisce una stringa ricercando un gruppo di caratteri che saranno immagazzinati in un vettore.
Vari	
`p`	Legge l'indirizzo di un puntatore avente lo stesso formato prodotto quando un indirizzo è inviato in output con `%p` in un'istruzione `printf`.
`n`	Immagazzina il numero dei caratteri presi in input fino a quel punto da `scanf`. L'argomento corrispondente è un puntatore a un intero.
`%`	Ignora un segno di percentuale (%) nell'input.

Figura 9.17 Gli indicatori di conversione per scanf

Il programma della Figura 9.18 leggerà degli interi, usando diversi indicatori di conversione per quel tipo di dato, e li visualizzerà come numeri decimali. Osservate che `%i` è in grado di ricevere in input degli interi decimali, ottali ed esadecimali.

```
1   /* Fig 9.18: fig09_18.c */
2   /* Leggere gli interi */
3   #include <stdio.h>
4
5   int main()
6   {
7     int a;
8     int b;
9     int c;
10    int d;
11    int e;
12    int f;
13    int g;
14
```

Figura 9.18 Leggere un input con gli indicatori di conversione per gli interi (continua)

```
15     printf("Enter seven integers: ");
16     scanf("%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g);
17
18     printf("The input displayed as decimal integers is:\n");
19     printf("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
20
21     return 0; /* indica che il programma è terminato con successo */
22
23  } /* fine della funzione main */
```

```
Enter seven integers: -70 -70 070 0x70 70 70 70
The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Figura 9.18 Leggere un input con gli indicatori di conversione per gli interi

Per prendere in input dei numeri in virgola mobile, potrà essere utilizzato uno qualsiasi degli indicatori di conversione per i valori in virgola mobile `e`, `E`, `f`, `g` o `G`. Il programma nella Figura 9.19 leggerà tre numeri in virgola mobile, con ognuno dei suddetti indicatori di conversione, e visualizzerà i tre numeri con la specifica di conversione `f`. Osservate che l'output del programma confermerà l'imprecisione dei valori in virgola mobile: questo aspetto sarà reso evidente dalla visualizzazione del terzo valore.

```
 1  /* Fig 9.19: fig09_19.c */
 2  /* Leggere dei numeri in virgola mobile */
 3  #include <stdio.h>
 4
 5  /* L'esecuzione del programma inizia dalla funzione main */
 6  int main()
 7  {
 8     double a;
 9     double b;
10     double c;
11
12     printf("Enter three floating-point numbers: \n");
13     scanf("%e%f%g", &a, &b, &c);
14
15     printf("Here are the numbers entered in plain\n");
16     printf("floating-point notation:\n");
17     printf("%f %f %f\n", a, b, c);
18
19     return 0; /* indica che il programma è terminato con successo */
20
21  } /* fine della funzione main */
```

Figura 9.19 Leggere un input con gli indicatori di conversione per i numeri in virgola mobile (continua)

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06
Here are the numbers entered in plain
floating-point notation:
1.279870
1279.870000
0.000003
```

Figura 9.19 Leggere un input con gli indicatori di conversione per i numeri in virgola mobile

I caratteri e le stringhe sono presi in input utilizzando rispettivamente gli indicatori di conversione c e s. Il programma della Figura 9.20 richiederà all'utente l'immissione di una stringa. Il programma prenderà in input il primo carattere della stringa con %c e lo immagazzinerà nella variabile di tipo carattere x. In seguito, prenderà in input il resto della stringa con %s e lo immagazzinerà nel vettore di caratteri y.

```
1   /* Fig 9.20: fig09_20.c */
2   /* Leggere i caratteri e le stringhe */
3   #include <stdio.h>
4
5   int main()
6   {
7     char x;
8     char y[ 9 ];
9
10    printf(«Enter a string: «);
11    scanf("%c%s", &x, y);
12
13    printf("The input was:\n");
14    printf("the character \"%c\" ", x);
15    printf("and the string \"%s\"\n", y);
16
17    return 0; /* indica che il programma è terminato con successo */
18
19  } /* fine della funzione main */
```

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

Figura 9.20 Prendere in input i caratteri e le stringhe

Una sequenza di caratteri potrà essere presa in input utilizzando un *gruppo di scansione*. In una stringa di controllo del formato, un gruppo di scansione è un insieme di caratteri racchiusi tra parentesi quadre [ ] e preceduti da un segno di percentuale. Un gruppo di scansione consentirà di esaminare i caratteri inseriti nello stream di input, selezionando solo

quelli che corrispondono ai caratteri contenuti nel gruppo di scansione. I caratteri che combaciano con quelli contenuti nel gruppo di scansione saranno immagazzinati nell'argomento corrispondente: un puntatore a un vettore di caratteri. Il gruppo di scansione fermerà l'input dei caratteri quando ne avrà incontrato uno che non sia contenuto nello stesso. Nel caso che il primo carattere letto dallo stream di input non corrisponda a uno di quelli contenuti nel gruppo di scansione, nel vettore sarà immagazzinato soltanto il carattere nullo. Il programma della Figura 9.21 utilizzerà il gruppo di scansione `[aeiou]` per scandire lo stream di input alla ricerca delle vocali. Osservate che, nell'esempio proposto, sono state lette solo le prime sette lettere. L'ottava lettera (`h`) non è inclusa nel gruppo di scansione e, di conseguenza, la scansione è stata interrotta.

```
1  /* Fig 9.21: fig09_21.c */
2  /* Usare un gruppo di scansione */
3  #include <stdio.h>
4
5  /* L'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8    char z[ 9 ];   /* dichiara il vettore z */
9
10   printf("Enter string: ");
11   scanf("%[aeiou]", z);
12
13   printf("The input was \"%s\"\n", z);
14
15   return 0; /* indica che il programma è terminato con successo */
16
17 } /* fine della funzione main */
```

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

Figura 9.21 Usare un gruppo di scansione

Il gruppo di scansione potrebbe essere utilizzato anche per selezionare dei caratteri diversi da quelli che esso contiene, utilizzando un *gruppo di scansione invertito*. Per creare un gruppo di scansione invertito inserirete un *caret* (`^`) all'interno delle parentesi quadre, prima dei caratteri di scansione. Ciò provocherà l'immagazzinamento dei caratteri che non sono inclusi nel gruppo di scansione. L'input sarà interrotto quando sarà stato incontrato uno dei caratteri contenuti nel gruppo di scansione invertito. Il programma della Figura 9.22 utilizzerà il gruppo di scansione invertito `[^aeiou]` per selezionare le consonanti o, detto in modo più appropriato, per selezionare le "non vocali".

Nella specifica di conversione di `scanf`, la dimensione di campo potrà essere utilizzata per leggere dallo stream di input un numero specificato di caratteri. Il programma della Figura 9.23 prenderà in input una serie di numeri consecutivi, suddividendoli in un primo intero di due cifre e in un secondo formato da quelle rimaste nello stream di input.

```
1  /* Fig 9.22: fig09_22.c */
2  /* Usare un gruppo di scansione invertito */
3  #include <stdio.h>
4
5  int main()
6  {
7    char z[ 9 ];
8
9    printf("Enter a string: ");
10   scanf("%[^aeiou]", z);
11
12   printf("The input was \"%s\"\n", z);
13
14   return 0; /* indica che il programma è terminato con successo */
15
16 } /* fine della funzione main */
```

```
Enter a string: String
The input was "Str"
```

Figura 9.22 Usare un gruppo di scansione invertito

```
1  /* Fig 9.23: fig09_23.c */
2  /* Prendere in input dei dati con una dimensione di campo */
3  #include <stdio.h>
4
5  int main()
6  {
7    int x;
8    int y;
9
10   printf(«Enter a six digit integer: «);
11   scanf(«%2d%d», &x, &y);
12
13   printf(«The integers input were %d and %d\n», x, y);
14
15   return 0; /* indica che il programma è terminato con successo */
16
17 } /* fine della funzione main */
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Figura 9.23 Prendere in input dei dati con una dimensione di campo

Spesso è necessario ignorare certi caratteri dello stream di input. Per esempio, una data potrebbe essere stata immessa come

```
11-10-1999
```

In questo caso ogni numero della data dovrà essere immagazzinato, mentre i trattini che li separano potranno essere ignorati. Per eliminare i caratteri non necessari, includeteli nella stringa di controllo del formato di `scanf` (i caratteri di spazio bianco, come lo spazio, il newline e la tabulazione, faranno ignorare tutti gli spazi bianchi iniziali). Per esempio, per ignorare i trattini nell'input, utilizzate l'istruzione

```
scanf("%d-%d-%d", &month, &day, &year);
```

Questa istruzione `scanf` elimina i trattini dall'input precedente, ma la data potrebbe anche essere immessa in un altro formato:

```
10/11/1999
```

In questo caso l'istruzione `scanf` precedente non eliminerebbe i caratteri inutili. Per questa ragione, `scanf` fornisce il *carattere di soppressione dell'assegnamento* `*`. Il carattere di soppressione dell'assegnamento consente a `scanf` l'input di un qualsiasi tipo di dato, che sarà però ignorato e non sarà assegnato ad alcuna variabile. Il programma della Figura 9.24 utilizzerà il carattere di soppressione dell'assegnamento nella specifica di conversione `%c`, per indicare che nello stream di input ci sarà un carattere che dovrà essere letto e ignorato. Di conseguenza, solo il mese, il giorno e l'anno saranno immagazzinati. I valori delle variabili saranno visualizzati per dimostrare che saranno stati presi in input nel modo corretto. Osservate che le liste degli argomenti di ognuna delle chiamate a `scanf` non contengono variabili corrispondenti alle specifiche di conversione che utilizzano il carattere di soppressione dell'assegnamento. I caratteri corrispondenti vengono semplicemente scartati.

```
1   /* Fig 9.24: fig09_24.c */
2   /* Leggere e ignorare i caratteri dello stream di input */
3   #include <stdio.h>
4
5   int main()
6   {
7     int month1;
8     int day1;
9     int year1;
10    int month2;
11    int day2;
12    int year2;
13
14    printf("Enter a date in the form mm-dd-yyyy: ");
15    scanf("%d%*c%d%*c%d", &month1, &day1, &year1);
16
17    printf("month = %d  day = %d  year = %d\n\n", month1, day1,
          year1);
18
19    printf("Enter a date in the form mm/dd/yyyy: ");
20    scanf("%d%*c%d%*c%d", &month2, &day2, &year2);
21
22    printf("month = %d  day = %d  year = %d\n", month2, day2,
          year2);
```

Figura 9.24 Leggere e ignorare i caratteri dello stream di input (continua)

```
23
24      return 0; /* indica che il programma è terminato con successo */
25
26   } /* fine della funzione main */
```

```
Enter a date in the form mm-dd-yyyy: 11-18-2003
month = 11  day = 18  year = 2003

Enter a date in the form mm/dd/yyyy: 11/18/2003
month = 11  day = 18  year = 2003
```

Figura 9.24 Leggere e ignorare i caratteri dello stream di input

Esercizi di autovalutazione

9.1 Riempite gli spazi bianchi in ognuna delle seguenti righe:

a) Tutto l'input e l'output è trattato in forma di _____.
b) Lo stream di _____ è connesso normalmente alla tastiera.
c) Lo stream di _____ è connesso normalmente con lo schermo del computer.
d) Una formattazione precisa dell'output può essere ottenuta con la funzione _____.
e) La stringa di controllo del formato può contenere _____, _____, _____ e _____.
f) Per inviare in output un intero decimale con segno, può essere utilizzato l'indicatore di conversione _____ o _____.
g) Gli indicatori di conversione _____ e _____ sono utilizzati per visualizzare degli interi senza segno rispettivamente in ottale, decimale ed esadecimale.
h) I modificatori _____ e _____ sono posti dinanzi agli indicatori di conversione per gli interi, per indicare che saranno visualizzati dei valori interi `short` o `long`.
i) L'indicatore di conversione _____ è utilizzato per visualizzare in notazione esponenziale un valore in virgola mobile.
j) Il modificatore _____ è posto dinanzi a un indicatore di conversione per numeri in virgola mobile per indicare che sarà visualizzato un valore `long double`.
k) Gli indicatori di conversione `e`, `E` e `f`, qualora non sia stata specificata una precisione, saranno visualizzati con _____ cifre a destra della virgola dei decimali.
l) Gli indicatori di conversione _____ e _____ sono utilizzati per visualizzare rispettivamente delle stringhe e dei caratteri.
m) Tutte le stringhe terminano con il carattere _____.
n) In una specifica di conversione per l'istruzione `printf`, la dimensione del campo e la precisione potranno essere controllate con un'espressione intera, inserendo al loro posto un _____ e ponendo nella lista degli argomenti una corrispondente espressione intera.
o) Il flag _____ provocherà l'allineamento a sinistra dei dati visualizzati in un campo.
p) Il flag _____ forzerà la visualizzazione del segno positivo o negativo.
q) Con la funzione _____ potrà essere ottenuta una formattazione precisa dell'input.
r) Un _____ è utilizzato per scandire una stringa alla ricerca di caratteri specifici per immagazzinarli in un vettore.
s) L'indicatore di conversione _____ potrà essere utilizzato per prendere in input degli interi con o senza segno espressi in ottale, decimale o esadecimale.
t) L'indicatore di conversione _____ potrà essere utilizzato per prendere in input un valore `double`.

u) Il _____ è utilizzato per leggere dei dati dallo stream di input e ignorarli senza assegnarli a una variabile.

v) Una _____ potrà essere utilizzata nella specifica di conversione di `scanf`, per indicare che dallo stream di input dovrà essere letto un numero specificato di caratteri o di cifre.

9.2 Trovate l'errore in ognuna delle seguenti istruzioni e spiegate come possa essere corretto.

a) La seguente istruzione dovrebbe visualizzare il carattere `'c'`
printf("%s\n", 'c');

b) La seguente istruzione dovrebbe visualizzare `9,375%`
printf("%.3f%", 9.375%);

c) La seguente istruzione dovrebbe visualizzare il primo carattere della stringa `"Monday"`
printf("%c\n", "Monday");

d) `printf(""A string in quotes"");`

e) `printf(%d%d, 12, 20);`

f) `printf("%c", "x");`

g) `printf("%s\n", 'Richard');`

9.3 Scrivete un'istruzione per ognuna delle seguenti attività:

a) Visualizzate `1234` giustificato a destra in un campo di `10` cifre.

b) Visualizzate `123,456789` in notazione esponenziale con un segno (+ o -) e `3` cifre di precisione.

c) Leggete un valore `double` nella variabile `number`.

d) Visualizzate `100` in formato ottale preceduto da `0`.

e) Leggete una stringa nel vettore di caratteri `string`.

f) Leggete dei caratteri nel vettore `n` finché non ne incontrate uno non numerico.

g) Utilizzate le variabili intere `x` e `y` per specificare la dimensione di campo e la precisione usate per visualizzare il valore `double 87,4573`.

h) Leggete un valore nel formato `3,5%`. Immagazzinate la percentuale nella variabile `float percent` ed eliminate il `%` dallo stream di input. Non usate il carattere di soppressione dell'assegnamento.

i) Visualizzate `3,333333` come un valore `long double` con un segno (+ o -) in un campo di `20` caratteri con una precisione di `3`.

Risposte agli esercizi di autovalutazione

9.1 a) Stream. b) Standard input. c) Standard output. d) `printf`. e) Indicatori di conversione, flag, dimensioni di campo, precisioni e caratteri letterali. f) `d`, `i`. g) `o`, `u`, `x` (o `X`). h) `h`, `l`. i) `e` (o `E`). j) `L`. k) 6. l) `s`, `c`. m) `NULL` (`'\0'`). n) asterisco (`*`). o) - (meno). p) + (più). q) `scanf`. r) Gruppo di scansione. s) `i`. t) `le`, `lE`, `lf`, `lg` o `lG`. u) Carattere di soppressione dell'assegnamento (`*`). v) Dimensione di campo.

9.2 a) Errore: l'indicatore di conversione `s` si attende un argomento di tipo puntatore a `char`.
Correzione: per visualizzare il carattere `'c'`, utilizzate la specifica di conversione `%c` o cambiate `'c'` in `"c"`.

b) Errore: tentativo di visualizzazione del carattere letterale `%` senza utilizzare la specifica di conversione `%%`.
Correzione: utilizzate `%%` per visualizzare un carattere `%` letterale.

c) Errore: l'indicatore di conversione `c` si attende un argomento di tipo `char`.
Correzione: per visualizzare il primo carattere di `"Monday"` utilizzate la specifica di conversione `%1s`.

d) Errore: tentativo di visualizzazione del carattere letterale `"` senza utilizzare la sequenza di escape `\"`.

Correzione: sostituite le virgolette più interne con `\"`.

e) Errore: la stringa di controllo del formato non è racchiusa tra virgolette.
Correzione: racchiudete `%d%d` tra virgolette.

f) Errore: il carattere `x` è racchiuso tra virgolette.
Correzione: perché possa essere visualizzata con `%c`, la costante di carattere dovrà essere racchiusa tra apici singoli.

g) Errore: la stringa da visualizzare è racchiusa tra apici singoli.
Correzione: per rappresentare una stringa, utilizzate le virgolette invece degli apici singoli.

9.3 a) `printf("%10d\n", 1234);`
b) `printf("%+.3e\n", 123.456789);`
c) `scanf("%lf", &number);`
d) `printf("%#o\n", 100);`
e) `scanf("%s", string);`
f) `scanf("%[0123456789]", n);`
g) `printf("%*.*f\n", x, y, 87.4573);`
h) `scanf("%f%%", &percent);`
i) `printf("%+20.3Lf\n", 3.333333);`

Esercizi

9.4 Scrivete un'istruzione `printf` o `scanf` per ognuna delle seguenti attività:

a) Visualizzate l'intero senza segno `40000` giustificato a sinistra in un campo di `15` cifre e con un minimo di `8` cifre.
b) Leggete un valore esadecimale nella variabile `hex`.
c) Visualizzate `200` con e senza segno.
d) Visualizzate `100` in formato esadecimale preceduto da `0x`.
e) Leggete dei caratteri nel vettore `s` finché non incontrate la lettera `p`.
f) Visualizzate `1,234` in un campo di `9` cifre preceduto da degli zeri.
g) Leggete un orario nel formato `hh:mm:ss` immagazzinando le sue parti nelle variabili intere `hour`, `minute` e `second`. Ignorate i due punti (`:`) inseriti nello stream di input. Utilizzate il carattere di soppressione dell'assegnamento.
h) Leggete dallo standard input una stringa nel formato `"characters"`. Immagazzinatela nel vettore di caratteri `s`. Eliminate le virgolette dallo stream di input.
i) Leggete un orario nel formato `hh:mm:ss` immagazzinando le sue parti nelle variabili intere `hour`, `minute` e `second`. Ignorate i due punti (`:`) inseriti nello stream di input. Non utilizzate il carattere di soppressione dell'assegnamento.

9.5 Mostrate che cosa sarà visualizzato da ognuna delle seguenti istruzioni. Nel caso che un'istruzione non sia corretta, indicatene il motivo.

a) `printf("%-10d\n", 10000);`
b) `printf("%c\n", "This is a string");`
c) `printf("%*.*lf\n", 8, 3, 1024.987654);`
d) `printf("%#o\n%#X\n%#e\n", 17, 17, 1008.83689);`
e) `printf("% ld\n%+ld\n", 1000000, 1000000);`
f) `printf("%10.2E\n", 444.93738);`
g) `printf("%19.2g\n", 444.93738);`
h) `printf("%d\n", 10.987);`

9.6 Trovate l'errore (o gli errori) in ognuno dei seguenti segmenti di programma. Per ognuno di essi, spiegate come possa essere corretto.

a) `printf("%s\n", 'Happy Birthday');`
b) `printf("%c\n", 'Hello');`
c) `printf("%c\n", "This is a string");`
d) L'istruzione successiva dovrebbe visualizzare `"Bon Voyage"`.
```
printf(""%s"", "Bon Voyage");
```
e)
```
char day[] = «Sunday»;
printf("%s\n", day[ 3 ]);
```
f) `printf('Enter your name: ');`
g) `printf(%f, 123.456);`
h) L'istruzione successiva dovrebbe visualizzare i caratteri `'O'` e `'K'`.
```
printf("%s%s\n", 'O', 'K');
```
i)
```
char s[ 10 ];
scanf("%c", s[ 7 ]);
```

9.7 Scrivete un programma che inizializzi i 10 elementi del vettore `number` con degli interi casuali compresi tra 1 e 1000. Visualizzate ogni valore e il totale progressivo del numero di caratteri visualizzati. Utilizzate la specifica di conversione `%n` per determinare il numero di caratteri inviati in output per ogni valore. Visualizzate il totale dei caratteri inviati in output per tutti i valori, includendo anche quello corrente, ogni volta che sarà visualizzato. L'output dovrà avere il seguente formato:

```
Value     Total characters
342       3
1000      7
963       10
6         11
ecc.
```

9.8 Scrivete un programma che verifichi la differenza tra gli indicatori di conversione `%d` e `%i` quando sono utilizzati nelle istruzioni `scanf`. Utilizzate le istruzioni

```
scanf("%i%d", &x, &y);
printf("%d %d\n", x, y);
```

per prendere in input e visualizzare i valori. Verificate il funzionamento del programma con i seguenti gruppi di dati:

```
  10    10
 -10   -10
 010   010
0x10  0x10
```

9.9 Scrivete un programma che visualizzi i valori di un puntatore utilizzando tutti gli indicatori di conversione per gli interi e la specifica `%p`. Quale visualizzerà dei valori strani? Quale provocherà degli errori? Sul vostro sistema, in quale formato visualizzerà l'indirizzo la specifica di conversione `%p`?

9.10 Scrivete un programma che verifichi i risultati ottenuti visualizzando il valore intero `12345` e quello in virgola mobile `1,2345`, in campi con varie dimensioni. Che cosa succederà quando i valori saranno visualizzati in campi con una dimensione inferiore a quelle dei valori?

9.11 Scrivete un programma che visualizzi il valore `100,453627` arrotondato all'intero più vicino e al numero più vicino a meno di un decimo, di un centesimo, di un millesimo e di un decimo di millesimo.

9.12 Scrivete un programma che prenda in input dalla tastiera una stringa e ne determini la lunghezza. Visualizzate la stringa utilizzando come dimensione di campo il doppio della sua lunghezza.

9.13 Scrivete un programma che converta degli interi, corrispondenti a temperature Fahrenheit comprese tra `0` e `212`, nelle equivalenti Celsius espresse in numeri a virgola mobile con `3` cifre di precisione. Utilizzate la formula

```
celsius = 5.0 / 9.0 * (fahrenheit - 32);
```

per eseguire il calcolo. L'output dovrà essere visualizzato giustificato a destra all'interno di due colonne, ognuna di 10 caratteri, e le temperature Celsius dovranno essere precedute da un segno sia per i valori positivi, sia per quelli negativi.

9.14 Scrivete un programma che verifichi tutte le sequenze di escape della Figura 9.16. Visualizzate un carattere prima e dopo le sequenze di escape che spostano il cursore, per rendere evidente lo spostamento del cursore.

9.15 Scrivete un programma che determini se il `?` possa essere visualizzato come carattere letterale, nella stringa di controllo del formato di `printf`, invece che con la sequenza di escape `\?`.

9.16 Scrivete un programma che prenda in input il valore `437` utilizzando con `scanf` ognuno degli indicatori di conversione per gli interi. Visualizzate ogni valore ricevuto in input utilizzando tutti gli indicatori di conversione per gli interi.

9.17 Scrivete un programma che utilizzi gli indicatori di conversione `e`, `f` e `g` per prendere in input il valore `1,2345`. Visualizzate i valori di ogni variabile per dimostrare che i suddetti indicatori di conversione potranno essere utilizzati per prendere in input lo stesso valore.

9.18 In alcuni linguaggi di programmazione, le stringhe devono essere immesse delimitandole con apici singoli o virgolette. Scrivete un programma che legga le tre stringhe `suzy`, `"suzy"` e `'suzy'`. Il C ignora gli apici singoli e le virgolette o li considera parti integranti della stringa?

9.19 Scrivete un programma che determini se, utilizzando l'indicatore di conversione `%c` nella stringa di controllo del formato di un'istruzione `printf`, il `?` possa essere visualizzato con la costante di carattere `'?'` invece che con la corrispondente sequenza di escape.

9.20 Scrivete un programma che utilizzi l'indicatore di conversione `g` per inviare in output il valore `9876,12345`. Visualizzate il valore con delle precisioni comprese tra `1` e `9`.

Capitolo 10

Le strutture, le unioni, la gestione dei bit e le enumerazioni in C

Obiettivi

- Essere in grado di creare e usare le strutture, le unioni e le enumerazioni.
- Essere in grado di passare le strutture per valore e per riferimento alle funzioni.
- Essere in grado di gestire i dati con gli operatori bitwise (orientati al bit).
- Essere in grado di creare campi di bit per immagazzinare i dati in modo compatto.

10.1 Introduzione

Le *strutture* sono collezioni di variabili correlate (a volte dette *aggregati*) sotto un unico nome. Le strutture possono contenere variabili di diversi tipi di dato (contrariamente ai vettori, che contengono soltanto elementi dello stesso tipo di dato). Le strutture sono usate comunemente per definire i record da salvare nei file (consultate il Capitolo 11, L'elaborazione dei file in C). I puntatori e le strutture facilitano la formazione di organizzazioni di dati più complesse come le liste concatenate, le code, le pile e gli alberi (consultate il Capitolo 12, Le strutture di dati in C).

10.2 La definizione delle strutture

Le strutture sono un *tipo di dato derivato* (sono cioè costruite usando oggetti di altri tipi). Considerate la seguente definizione di struttura:

```
struct card {
   char *face;
   char *suit;
};
```

La parola chiave `struct` introduce la definizione della struttura. L'identificatore `card` è la structure tag (etichetta della struttura), che attribuisce un nome alla definizione della struttura ed è usata con la parola chiave `struct`, per dichiarare le variabili di quel *tipo di struttura*. In questo esempio, il tipo di struttura è `struct card`. Le variabili dichiarate all'interno delle parentesi graffe della definizione sono i *membri* della struttura. I membri dello stesso tipo di struttura devono avere nomi univoci, mentre due diverse strutture potrebbero contenere membri con lo stesso nome, senza creare conflitti (vedremo presto perché). Ogni definizione di struttura deve terminare con un punto e virgola.

Errore tipico 10.1

Dimenticare il punto e virgola, al termine della definizione della struttura è un errore di sintassi.

La definizione di `struct card` contiene due membri di tipo `char *`: `face` e `suit`. I membri delle strutture potrebbero essere variabili di un tipo fondamentale di dato (per esempio: `int`, `float` ecc.), o aggregati, come i vettori e altre strutture. Da quanto abbiamo visto nel Capitolo 6, ogni elemento di un vettore deve essere dello stesso tipo. I membri delle strutture, invece, possono essere di tipi di dato diversi. Per esempio,

```
struct employee {
      char firstName [ 20 ];
      char lastName [ 20 ];
      int age;
      char gender;
      double hourlySalary;
};
```

contiene membri vettori di caratteri per il nome e il cognome, un membro di tipo `int` per l'età dell'impiegato, uno di tipo `char` contenente `'M'` o `'F'` per il suo sesso e uno di tipo `double` per il suo salario orario. Una struttura non può contenere un'istanza di se stessa. Una variabile di tipo `struct card`, per esempio, non può essere dichiarata all'interno della definizione della `struct employee`. Può però essere incluso un puntatore a `struct employee`. Per esempio,

```
struct employee2 {
      char firstName [ 20 ];
      char lastName [ 20 ];
      int age;
      char gender;
      double hourlySalary;
      struct employee2 person;  /* ERRORE */
      struct employee2 *ePtr;     /* puntatore */
};
```

`struct employee2` contiene un'istanza di se stessa (`person`), che costituisce un errore. Siccome `ePtr` è un puntatore (al tipo `struct employee2`), è permesso includerlo nella definizione. Una struttura che contenga un membro che è un puntatore allo stesso tipo di struttura è detta *struttura ricorsiva*. Questa sarà usata nel Capitolo 12 per costruire vari tipi di strutture di dati collegate.

Le definizioni di struttura non riservano alcuno spazio in memoria, ma ognuna di esse crea piuttosto un nuovo tipo di dato che sarà usato per dichiarare le variabili. Le variabili di tipo struttura sono dichiarate allo stesso modo delle variabili di altro tipo. La dichiarazione

```
struct card aCard, deck[ 52 ], *cardPtr;
```

dichiarerà `aCard` come una variabile di tipo `struct card`, `deck` come un vettore di 52 elementi di tipo `struct card` e `cardPtr` come un puntatore a una `struct card`. Le variabili di un dato tipo di struttura possono anche essere dichiarate contestualmente alla sua definizione, inserendo un elenco di nomi di variabile separati da virgole, tra la parentesi graffa chiusa della definizione di struttura e il punto e virgola che chiude la stessa. Per esem-

pio, la dichiarazione precedente potrebbe essere incorporata nella definizione della struttura `struct card` in questo modo:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Il nome della structure tag è opzionale. Nel caso in cui una definizione di struttura non contenga un nome di structure tag, le variabili di quel tipo di struttura non potranno essere dichiarate separatamente, ma solo contestualmente alla definizione della struttura.

Buona abitudine 10.1

Fornite un nome di structure tag quando create un tipo di struttura. Il nome di structure tag è un modo conveniente per dichiarare nuove variabili di quel tipo di struttura, nel resto del programma.

Buona abitudine 10.2

Scegliere un nome di structure tag significativo aiuta a rendere il programma auto-esplicativo.

Le uniche operazioni valide effettuabili con le strutture sono le seguenti: assegnare variabili di struttura a variabili dello stesso tipo di struttura, rilevare l'indirizzo (`&`) di una variabile di tipo struttura, accedere ai suoi membri (consultate la Sezione 10.4) e usare l'operatore `sizeof`, per determinarne la dimensione.

Errore tipico 10.2

Assegnare una struttura a una di tipo differente genera un errore di compilazione.

Le strutture non possono essere confrontate utilizzando gli operatori `==` e `!=`, perché i membri della stessa non sono necessariamente immagazzinati in byte di memoria consecutivi. A volte in una struttura potrebbero esserci dei "buchi", dovuti al fatto che i computer possono immagazzinare certi tipi di dato soltanto entro confini di memoria ben determinati come la mezza parola (halfword), la parola (word), o la doppia parola (doubleword). Una parola è un'unità standard di memoria, formata solitamente da due o quattro byte, usata per memorizzare i dati in un computer. Considerate la seguente definizione di struttura, in cui `sample1` e `sample2` saranno dichiarate di tipo `struct example`:

```
struct example {
   char c;
   int i;
} sample1, sample2;
```

Un computer con una parola di due byte potrebbe richiedere l'allineamento con i confini della parola, ovverosia con l'inizio della stessa, per ognuno dei membri della `struct example` (tutto ciò dipenderà dalla macchina). La Figura 10.1 mostra un esempio di allineamento in memoria, per una variabile di tipo `struct example` ai cui membri siano stati assegnati il carattere `'a'` e l'intero `97`; nell'esempio è mostrata la rappresentazione in bit dei valori. Nel caso in cui i membri fossero memorizzati a cominciare dai limiti della parola, nell'immagazzinamento delle variabili del tipo `struct example`, si creerebbe un buco di un byte (il numero uno, nella figura). Il valore immagazzinato in quel buco di un byte sarebbe indefinito.

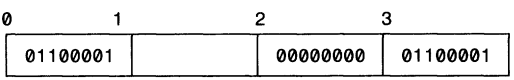


Figura 10.1 Un possibile allineamento di memorizzazione per una variabile di tipo struct example mostra un'area indefinita nella memoria.

Un confronto delle due strutture potrebbe non considerarle identiche, neanche nel caso in cui i valori dei membri `sample1` e `sample2` fossero effettivamente uguali, perché sarebbe molto improbabile che i valori indefiniti, contenuti nei due buchi di un byte, avessero lo stesso contenuto.

Errore tipico 10.3

Confrontare le strutture è un errore di sintassi.

Obiettivo portabilità 10.1

La rappresentazione di una struttura dipende dalla macchina, così come la dimensione dei dati di un particolare tipo e le considerazioni sull'allineamento in memoria.

10.3 Inizializzare le strutture

Le strutture possono essere inizializzate, allo stesso modo dei vettori, usando degli elenchi di inizializzazione. Per inizializzare una struttura immettete nella sua dichiarazione, dopo il nome della variabile, un segno di uguale e un elenco di inizializzatori, racchiusi tra parentesi graffe e separati da virgole. Per esempio, la dichiarazione

```
struct card aCard = { "Three", "Hearts" };
```

creerà la variabile `aCard` di tipo `struct card` (come definita nella Sezione 10.2) e inizializzerà il membro `face` con "`Three`" e `suit` con "`Hearts`". Nel caso in cui nell'elenco ci fossero meno valori di inizializzazione di quanti siano i membri della struttura, quelli rimanenti sarebbero inizializzati automaticamente con `0` (o `NULL`, qualora il membro sia un puntatore). Le variabili di struttura dichiarate all'esterno di una definizione di funzione saranno inizializzate con `0` o `NULL`, qualora non siano state inizializzate in modo esplicito contestualmente alla loro dichiarazione. Le variabili di struttura potrebbero anche essere inizializzate con istruzioni di assegnamento, utilizzando una variabile di struttura dello stesso tipo, o assegnando dei valori ai singoli membri della struttura.

10.4 Accedere ai membri delle strutture

Per accedere ai membri delle strutture, potrete usare due operatori: l'*operatore membro di struttura* (`.`) (detto anche *operatore punto*) e l'*operatore puntatore a struttura* (`->`) (detto anche *operatore freccia*). L'operatore membro di struttura accede a un membro della stessa attraverso il nome della variabile di struttura. Per esempio, per stampare il membro `suit` della variabile di tipo struttura `aCard` dichiarata nella Sezione 10.3, si userà l'istruzione

```
printf( "%s", aCard.suit ); /* visualizza Hearts */
```

L'operatore puntatore a struttura (che consiste in un segno meno (-) e in un segno di maggiore (>), senza spazi interposti) accede ai membri di una struttura, attraverso un *puntatore alla*

stessa. Supponete che sia stato dichiarato il puntatore `cardPtr`, che sia stato fatto puntare a `struct card` e che l'indirizzo della struttura `aCard` sia stato assegnato a `cardPtr`. Per stampare il membro `suit` della struttura `aCard` con il puntatore `cardPtr`, potrete usare l'istruzione

```
printf( "%s", cardPtr->suit ); /* visualizza Hearts */
```

L'espressione `cardPtr->suit` è equivalente a `(*cardPtr).suit`, che risolve il riferimento del puntatore e accede al membro `suit`, usando l'operatore membro di struttura. In questo caso, le parentesi sono necessarie perché l'operatore membro di struttura (`.`) ha una priorità più alta dell'operatore di risoluzione del riferimento (`*`). Gli operatori puntatore a struttura e membro di struttura, insieme alle parentesi tonde (per le chiamate di funzioni) e a quelle quadre (`[ ]`) usate per gli indici dei vettori, hanno la priorità più alta tra gli operatori e associano da sinistra a destra.

Collaudo e messa a punto 10.1

Evitate di usare gli stessi nomi per i membri appartenenti a tipi di struttura differenti. È consentito, ma potrebbe causare confusione.

Buona abitudine 10.3

Non inserite degli spazi intorno agli operatori "->" e ". ". Omettere gli spazi aiuterà a evidenziare che le espressioni in cui quegli operatori sono contenuti sono essenzialmente nomi di una singola variabile.

Errore tipico 10.4

Inserire degli spazi tra i caratteri "-" e ">" dell'operatore puntatore a struttura (così come inserirli tra quelli di qualsiasi altro operatore formato da più caratteri, a eccezione di "?:") è un errore di sintassi.

Errore tipico 10.5

Tentare di fare riferimento a un membro di una struttura, usando soltanto il nome del membro è un errore di sintassi.

Errore tipico 10.6

Non usare le parentesi qualora si punti a un membro di struttura, usando un puntatore e l'operatore membro di struttura (per esempio, `*cardPtr.suit`) è un errore di sintassi.

Il programma della Figura 10.2 mostrerà l'uso degli operatori membro di struttura e puntatore a struttura. Usando l'operatore membro di struttura, ai membri di `aCard` saranno assegnati rispettivamente i valori "`Ace`" e "`Spades`" (righe 18 e 19). L'indirizzo della struttura `aCard` sarà assegnato al puntatore `cardPtr` (riga 21). La funzione `printf` visualizzerà i membri della variabile `aCard`, usando l'operatore membro di struttura con il nome di variabile `aCard`, l'operatore puntatore a struttura con `cardPtr`, e l'operatore membro di struttura con `cardPtr`, ovviamente dopo averne risolto il riferimento (dalla riga 23 alla riga 25).

```
1  /* Fig. 10.2: fig10_02.c
2     Usare gli operatori membro di struttura
3     e puntatore a struttura */
4  #include <stdio.h>
5
6  /* dichiarazione della struttura card */
7  struct card {
8     char *face;  /* dichiara un puntatore alla figura */
9     char *suit;   /* dichiara un puntatore al seme */
10 }; /* fine della struttura card */
11
12 int main()
13 {
14    struct card aCard; /* dichiara una variabile di tipo
          struct card */
15    struct card *cardPtr; /* dichiara un puntatore a una
          struct card */
16
17    /* inserisce delle stringhe in aCard */
18    aCard.face = "Ace";
19    aCard.suit = "Spades";
20
21    cardPtr = &aCard; /* assegna l'indirizzo di aCard a cardPtr */
22
23    printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ",
          aCard.suit,
24           cardPtr->face, " of ", cardPtr->suit,
25           (*cardPtr).face, " of ", (*cardPtr).suit);
26
27    return 0; /* indica che il programma è terminato con successo */
28
29 } /* fine della funzione main */
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Figura 10.2 Usare l'operatore membro di struttura e l'operatore puntatore a struttura

10.5 Usare le strutture con le funzioni

Le strutture possono essere passate alle funzioni, fornendo i singoli membri, l'intera struttura o un puntatore alla stessa. Nel caso in cui si fornisca a una funzione una struttura o i suoi singoli membri, questi saranno passati per valore. Di conseguenza, i membri della struttura dichiarati nella funzione chiamante non potranno essere modificati da quella chiamata.

Per passare una struttura per riferimento, dovrete passare alla funzione l'indirizzo della variabile di struttura. Al pari di tutte le altre, i vettori di strutture sono passati automaticamente per riferimento.

Nel Capitolo 6, abbiamo asserito che un vettore potrà essere passato per valore mediante l'uso di una struttura. Per passare un vettore per valore, dovrete creare una struttura il cui membro sia il vettore. Le strutture sono passate per valore, di conseguenza il vettore sarà passato nello stesso modo.

Errore tipico 10.7

Ritenere che, in una chiamata di funzione, le strutture siano passate automaticamente per riferimento, come avviene per i vettori, e tentare di modificare i valori contenuti nella struttura del chiamante all'interno della funzione chiamata è un errore logico.

Obiettivo efficienza 10.1

Il passaggio per riferimento delle strutture, in una chiamata a funzione, è un modo più efficiente del passaggio per valore (poiché il secondo metodo richiede una copia dell'intera struttura).

10.6 typedef

La parola chiave `typedef` fornisce un meccanismo per creare dei sinonimi, o pseudonimi, per i tipi di dato definiti in precedenza. Per abbreviare i nomi dei tipi di struttura, questi sono spesso definiti con `typedef`. Per esempio, l'istruzione

```
typedef struct card Card;
```

definirà il nuovo nome di tipo `Card` come un sinonimo per il tipo `struct card`. Il programmatore C usa spesso `typedef` per definire un tipo di struttura, in modo da evitare l'uso della structure tag. Per esempio, la definizione seguente

```
typedef struct {
   char *face;
   char *suit;
} Card;
```

creerà il tipo di struttura `Card`, senza la necessità di un'istruzione `typedef` separata.

Buona abitudine 10.4

Scrivete in maiuscolo la prima lettera dei nomi definiti con `typedef`*, in modo da rendere evidente che sono dei sinonimi per i nomi di altri tipi di dato.*

`Card` a questo punto potrà essere usata per dichiarare delle variabili di tipo `struct card`. La dichiarazione

```
Card deck[ 52 ];
```

dichiarerà un vettore di 52 strutture `Card` (ovverosia variabili di tipo `struct card`). Creare un nuovo nome con `typedef` non significa creare un altro tipo; `typedef` crea semplicemente un nuovo nome di tipo, che potrà essere usato come pseudonimo per quello di un tipo già esistente. Un nome significativo aiuterà a rendere il programma auto-esplicativo. Per esempio, leggendo la dichiarazione precedente si saprà che "`deck` è un vettore di 52 `Card`".

Spesso `typedef` è usato per creare sinonimi per i tipi di dato fondamentali. Un programma che richiedesse un intero di quattro byte, per esempio, potrebbe usare un `int` su un sistema e un `long` su un altro. I programmi progettati per essere portabili useranno spesso

`typedef` per creare uno pseudonimo, come `Integer` per gli interi di quattro byte. Nel programma, basterà modificare la sola definizione dello pseudonimo `Integer`, per farlo funzionare su entrambi i sistemi.

Obiettivo portabilità 10.2

Usare `typedef`, *per aiutare a rendere più portabile il programma.*

10.7 Esempio: simulazione di un mescolatore e distributore di carte ad alta efficienza

Il programma della Figura 10.3 è basato sulla simulazione di un mescolatore e distributore di carte discussa nel Capitolo 7. Il programma rappresenta il mazzo di carte in un vettore di strutture. Per mescolare e distribuire le carte, il programma utilizza degli algoritmi ad alta efficienza. Nella Figura 10.4 è mostrato l'output del programma.

All'interno del programma, la funzione `fillDeck` (righe 44-55) inizializzerà il vettore di tipo `Card` inserendo le carte, in modo ordinato, dall'Asso al Re di ogni seme. Il vettore di tipo `Card` sarà quindi passato (nella riga 36) alla funzione `shuffle` (righe 58-72), in cui è implementato l'algoritmo di mescolamento ad alta efficienza. La funzione `shuffle` riceverà come argomento un vettore di 52 strutture di tipo `Card`. La funzione scorrerà le 52 carte (gli indici di vettore compresi tra zero e 51) usando un comando `for` nelle righe 65-70. Per ogni carta, sarà scelto un numero casuale compreso tra zero e 51. In seguito, la struttura `Card` corrente sarà scambiata all'interno del vettore con quella selezionata casualmente (dalla riga 67 alla riga 69). In un singolo passaggio sull'intero vettore di strutture `Card` si eseguiranno 52 scambi in tutto e, alla fine, questo risulterà mescolato! Questo algoritmo di mescolamento non può dar luogo a un differimento indefinito, come quello presentato nel Capitolo 7. Per distribuire le carte mescolate, l'algoritmo di distribuzione ad alta efficienza implementato nella funzione `deal` (righe 75-85) richiederà soltanto un passaggio sul vettore, poiché le strutture `Card` sono già state scambiate di posto all'interno dello stesso.

Errore tipico 10.8

Dimenticare di includere l'indice di vettore, allorquando si faccia riferimento alle singole strutture di un vettore è un errore di sintassi.

```
1   /* Fig. 10.3: fig10_03.c
2      L'utilizzo delle strutture nel programma per mescolare
        e distribuire le carte */
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <time.h>
6
7   /* dichiarazione della struttura card */
8   struct card {
9      char *face;  /* dichiara un puntatore alla figura */
10     char *suit;   /* dichiara un puntatore al seme */
```

Figura 10.3 Simulazione di un mescolatore e distributore di carte ad alta efficienza (continua)

```
11  }; /* fine della struttura card */
12
13  typedef struct card Card; /* nuovo nome di tipo per struct card */
14
15  /* prototipi */
16  void fillDeck( Card * const wDeck, const char * wFace[],
17       const char * wSuit[] );
18  void shuffle( Card * const wDeck );
19  void deal( const Card * const wDeck );
20
21  int main()
22  {
23     Card deck[52]; /* dichiara il vettore di elementi di tipo Card */
24
25     /* inizializza il vettore di puntatori */
26     const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
27                    "Six", "Seven", "Eight", "Nine", "Ten",
28                    "Jack", "Queen", "King" };
29
30     /* inizializza il vettore di puntatori */
31     const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades" };
32
33     srand( time ( NULL ) ); /* randomizzazione */
34
35     fillDeck( deck, face, suit ); /* carica il vettore deck
                                      con gli elementi di tipo Card */
36     shuffle( deck );      /* sistema gli elementi di tipo Card
                                 in ordine casuale */
37     deal( deck );         /* distribuisce tutti i 52 elementi
                                 di tipo Card */
38
39     return 0; /* indica che il programma è terminato con successo */
40
41  } /* fine della funzione main */
42
43  /* inserisce le stringhe nelle strutture di tipo Card */
44  void fillDeck( Card * const wDeck, const char *wFace[],
45       const char *wSuit[] )
46  {
47     int i; /* contatore */
48
49     /* itera scorrendo gli elementi di wDeck */
50     for ( i = 0; i <= 51; i++ ) {
51        wDeck[ i ].face = wFace[ i % 13 ];
52        wDeck[ i ].suit = wSuit[ i / 13 ];
53     } /* fine del comando for */
54
55  } /* fine della funzione fillDeck */
56
```

Figura 10.3 Simulazione di un mescolatore e distributore di carte ad alta efficienza (continua)

```
57  /* mescola le carte */
58  void shuffle( Card * const wDeck )
59  {
60     int i;    /* contatore */
61     int j;    /* variabile contenente un valore casuale compreso
                    tra 0 e 51 */
62     Card temp; /* dichiara una struttura temporanea per scambiare
                     elementi di tipo Card */
63
64     /* itera scorrendo wDeck e scambiando casualmente gli elementi
          di tipo Card */
65     for ( i = 0; i <= 51; i++ ) {
66        j = rand() % 52;
67        temp = wDeck[ i ];
68        wDeck[ i ] = wDeck[ j ];
69        wDeck[ j ] = temp;
70     } /* fine del comando for */
71
72  } /* fine della funzione shuffle */
73
74  /* distribuisce le carte */
75  void deal( const Card * const wDeck )
76  {
77     int i; /* contatore */
78
79     /* itera scorrendo gli elementi di wDeck */
80     for ( i = 0; i <= 51; i++ ) {
81        printf("%5s of %-8s%c", wDeck[ i ].face, wDeck[ i ].suit,
82               ( i + 1 ) % 2 ? '\t' : '\n');
83     } /* fine del comando for */
84
85  } /* fine della funzione deal */
```

Figura 10.3 Simulazione di un mescolatore e distributore di carte ad alta efficienza

```
 Four of Clubs          Three of Hearts
Three of Diamonds       Three of Spades
 Four of Diamonds         Ace of Diamonds
 Nine of Hearts           Ten of Clubs
Three of Clubs           Four of Hearts
Eight of Clubs           Nine of Diamonds
Deuce of Clubs          Queen of Clubs
Seven of Clubs           Jack of Spades
  Ace of Clubs           Five of Diamonds
  Ace of Spades          Five of Clubs
Seven of Diamonds         Six of Spades
Eight of Spades         Queen of Hearts
 Five of Spades         Deuce of Diamonds
```

Figura 10.4 L'output della simulazione di un mescolatore e distributore di carte ad alta efficienza (continua)

```
Queen of Spades          Six of Hearts
Queen of Diamonds      Seven of Hearts
  Jack of Diamonds     Nine of Spades
 Eight of Hearts       Five of Hearts
  King of Spades        Six of Clubs
 Eight of Diamonds      Ten of Spades
   Ace of Hearts       King of Hearts
  Four of Spades       Jack of Hearts
 Deuce of Hearts       Jack of Clubs
 Deuce of Spades        Ten of Diamonds
 Seven of Spades       Nine of Clubs
  King of Clubs         Six of Diamonds
   Ten of Hearts       King of Diamonds
```

Figura 10.4 L'output della simulazione di un mescolatore e distributore di carte ad alta efficienza

10.8 Le unioni

Un'*unione*, come una struttura, è un tipo di dato derivato i cui membri condividono lo stesso spazio di memoria. In varie situazioni in un programma, alcune variabili potrebbero essere irrilevanti mentre altre potrebbero non esserlo; proprio per questo motivo, un'unione condivide lo spazio invece di sprecare la memoria per variabili che non siano utilizzate. I membri di un'unione possono essere di qualsiasi tipo di dato. Il numero di byte utilizzato per memorizzare un'unione deve essere sufficiente a contenere il membro più grande. Nella maggior parte dei casi, le unioni conterranno due o più tipi di dato, ma si potrà fare riferimento a un solo membro alla volta e, quindi, a un solo tipo di dato. Sarà responsabilità del programmatore assicurarsi che si faccia riferimento ai dati di un'unione con il tipo di dato appropriato.

Errore tipico 10.9

È un errore logico far riferimento a un membro all'interno di un'unione con una variabile di tipo inadeguato.

Obiettivo portabilità 10.3

Qualora abbiate fatto riferimento a un dato immagazzinato in un'unione usando un tipo di dato diverso da quello della sua dichiarazione, i risultati che otterrete dipenderanno dall'implementazione.

Un'unione è dichiarata con la parola chiave `union` usando un formato simile a quello della dichiarazione delle strutture. La dichiarazione della `union`

```
union number {
   int x;
   double y;
};
```

indica che `number` sarà un tipo `union` e che i suoi membri saranno `int x` e `double y`. Normalmente la definizione di un'unione sarà inserita in un file header e inclusa in tutti i file sorgente che utilizzino quel tipo unione.

Ingegneria del software 10.1

Al pari di una dichiarazione `struct`, *quella della* `union` *crea semplicemente un nuovo tipo di dato. Posizionare una dichiarazione* `union` *o* `struct` *al di fuori di qualsiasi funzione non crea dunque una variabile globale.*

Le operazioni che possono essere eseguite con un'unione sono le seguenti: assegnare un'unione a un'altra dello stesso tipo, rilevare l'indirizzo (&) di una variabile di tipo unione e accedere ai membri di un'unione, usando gli operatori membro di struttura e puntatore a struttura. Le unioni non possono essere confrontate utilizzando gli operatori == e !=, per le stesse ragioni per cui non possono esserlo le strutture.

In una dichiarazione, un'unione può essere inizializzata con un valore dello stesso tipo del suo primo membro. Per esempio, con l'unione precedente, la dichiarazione

```
union number value = { 10 };
```

sarebbe una valida inizializzazione per la variabile `value` dell'unione, perché questa sarebbe inizializzata con un `int`, mentre la seguente dichiarazione troncherebbe la parte decimale del valore usato per l'inizializzazione e normalmente produrrebbe un messaggio di avvertimento da parte del compilatore:

```
union number value = { 1.43 };
```

Errore tipico 10.10

Confrontare le unioni è un errore di sintassi.

Obiettivo portabilità 10.4

La quantità di memoria richiesta per memorizzare un'unione dipende dall'implementazione.

Obiettivo portabilità 10.5

Alcune unioni potrebbero non essere facilmente portabili su altri sistemi. La portabilità di un'unione dipende spesso dalle necessità di allineamento in memoria, su un dato sistema, per i tipi di dato utilizzati dai membri dell'unione.

Obiettivo efficienza 10.2

Le unioni risparmiano la memoria.

Il programma della Figura 10.3 userà la variabile `value` (riga 13) di tipo `union number`, per visualizzare il valore immagazzinato nell'unione sia come `int` e sia come `double`. L'output del programma dipenderà dall'implementazione. Lo stesso output dimostrerà che la rappresentazione interna di un valore `double` potrebbe essere sensibilmente diversa da quella di un `int`.

```
1  /* Fig. 10.5: fig10_05.c
2     Un esempio di unione */
3  #include <stdio.h>
4
```

Figura 10.5 Stampare i valori di un'unione con i tipi di dato di entrambi i membri (continua)

```
 5  /* dichiarazione dell'unione number */
 6  union number {
 7     int x;
 8     double y;
 9  }; /* fine dell'unione number */
10
11  int main()
12  {
13    union number value;    /* dichiarazione di una variabile
                                di tipo unione */
14
15    value.x = 100; /* memorizza un intero nell'unione */
16    printf("%s\n%s\n%s%d\n%s%f\n\n",
17           "Put a value in the integer member",
18           "and print both members.",
19           "int:   ", value.x,
20           "float: ", value.y);
21
22    value.y = 100.0; /* memorizza un valore di tipo double
                          nella stessa unione */
23    printf("%s\n%s\n%s%d\n%s%f\n",
24           "Put a value in the floating member",
25           "and print both members.",
26           "int:   ", value.x,
27           "float: ", value.y);
28
29    return 0; /* indica che il programma è terminato con successo */
30
31  } /* fine della funzione main */
```

```
Put a value in the integer member
and print both members.
int:   100
double:
-92559592117433136000000000000000000000000000000000000000000000000.000000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

Figura 10.5 Stampare i valori di un'unione con i tipi di dato di entrambi i membri

10.9 Gli operatori bitwise

Tutti i dati sono rappresentati all'interno dei computer come sequenze di bit. Ogni bit può assumere i valori 0 o 1. Sulla maggior parte dei sistemi, una sequenza di otto bit forma un byte (l'unità standard di memorizzazione per una variabile di tipo `char`). Gli altri tipi di dato sono memorizzati in un numero maggiore di byte. Gli operatori bitwise (orientati al bit)

sono usati per gestire i bit dei loro operandi interi (`char`, `short`, `int` e `long`; `signed` o `unsigned`). Con gli operatori bitwise si usano normalmente gli interi senza segno.

Obiettivo portabilità 10.6

I risultati delle manipolazioni bitwise dei dati dipendono dalla macchina.

Osservate che nella trattazione sugli operatori bitwise di questa sezione saranno mostrate le rappresentazioni binarie degli operandi interi. Per una spiegazione dettagliata del sistema numerico binario (detto anche "sistema numerico in base 2"), consultate l'Appendice E, I sistemi numerici. Inoltre, i programmi inclusi nelle Sezioni 10.9 e 10.10 sono stati provati utilizzando Microsoft Visual C++. Questi programmi potrebbero anche non funzionare sul vostro sistema, a causa della dipendenza dalla macchina delle manipolazioni bitwise.

Gli operatori bitwise sono: *AND bitwise* (&), OR *inclusivo bitwise* (|), *OR esclusivo bitwise* (^), *scorrimento a sinistra* (<<), *scorrimento a destra* (>>), e *complemento* (~). Gli operatori bitwise AND, OR inclusivo e OR esclusivo confrontano ogni singolo bit dei loro due operandi. L'operatore bitwise AND imposta a 1 ogni bit del risultato, qualora entrambi quelli corrispondenti nei suoi due operandi siano `1`. L'operatore bitwise OR inclusivo imposta a `1` ogni bit del risultato, qualora almeno uno (o entrambi) di quelli corrispondenti nei suoi due operandi sia `1`. L'operatore bitwise OR esclusivo imposta a 1 ogni bit del risultato, qualora solo uno di quelli corrispondenti nei suoi due operandi sia 1. L'operatore di scorrimento a sinistra fa scorrere in quella direzione i bit del suo operando di sinistra, per un numero di volte specificato da quello di destra. L'operatore di scorrimento a destra fa scorrere in quella direzione i bit del suo operando di sinistra, per un numero di volte specificato da quello di destra. L'operatore bitwise complemento imposta a `1` i bit del risultato, qualora quelli corrispondenti nel suo operando siano impostati a `0`, mentre imposta a `0` i bit del risultato, qualora quelli corrispondenti nel suo operando siano impostati a `1`. Potrete trovare una discussione dettagliata di ogni operatore bitwise negli esempi che seguiranno. Gli operatori bitwise sono riepilogati nella Figura 10.6.

Operatore	Descrizione
`&` AND bitwise	I bit del risultato sono impostati a `1`, se quelli corrispondenti nei due operandi sono entrambi `1`.
`|` OR inclusivo bitwise	I bit del risultato sono impostati a `1`, se almeno uno dei bit corrispondenti nei due operandi è `1`.
`^` OR esclusivo bitwise	I bit del risultato sono impostati a `1`, se solo uno dei bit corrispondenti nei due operandi è `1`.
`<<` scorrimento a sinistra	Fa scorrere a sinistra i bit del primo operando, per un numero di volte specificato dal secondo operando; i bit rimasti vuoti a destra sono riempiti con `0`.
`>>` scorrimento a destra	Fa scorrere a destra i bit del primo operando, per un numero di volte specificato dal secondo operando; il metodo di riempimento dei bit rimasti vuoti a sinistra dipende dalla macchina.
`~` complemento a uno	Tutti i bit a `0` sono impostati a `1` e tutti quelli a `1` sono impostati a `0`.

Figura 10.6 Gli operatori bitwise

Visualizzare un intero senza segno come sequenza di bit

Quando si utilizzano gli operatori bitwise, è utile visualizzare i valori nella loro rappresentazione binaria, in modo da mostrare precisamente l'effetto dei suddetti operatori. Il programma della Figura 10.7 visualizzerà un intero `unsigned` nella sua rappresentazione binaria, suddividendola in gruppi di otto bit. La funzione `displayBits` (righe 21-42) userà l'operatore bitwise AND per combinare le variabili `value` e `displayMask` (riga 32). L'operatore bitwise AND è usato spesso con un operando *maschera*: un valore intero con specifici bit impostati a `1`. Le maschere sono usate per nascondere alcuni bit di un valore, mentre si selezionano gli altri. Nella funzione `displayBits`, alla variabile maschera `displayMask` sarà assegnato il valore

```
1 << 31 (10000000 00000000 00000000 00000000)
```

L'operatore di scorrimento a sinistra farà scorrere il valore `1` dal bit meno significativo (ovverosia, quello più a destra) di `displayMask` a quello più significativo (ovverosia, quello più a sinistra) e riempirà i bit rimasti vuoti a destra con `0`. La riga 32

```
putchar( value & displayMask ? '1' : '0' );
```

determinerà se per il bit più a sinistra della variabile `value` dovrà essere visualizzato un `1` o uno `0`. Nel momento in cui `value` e `displayMask` saranno combinate usando `&`, tutti i bit della variabile `value`, eccettuato quello più significativo, saranno "mascherati" (nascosti) poiché ogni bit "messo in AND" con `0` produce `0`. Qualora il bit più a sinistra sia un `1`, `value & displayMask` sarà un valore diverso da 0 e sarà quindi visualizzato un `1`, altrimenti sarà stampato uno `0`. La variabile `value` sarà quindi fatta scorrere a sinistra di un bit dall'espressione `value <<= 1` (che è equivalente a `value = value << 1`). I suddetti passi saranno ripetuti per ogni bit della variabile `unsigned value`.

```
1  /* Fig. 10.7: fig10_07.c
2     Visualizzare in bit un intero senza segno */
3  #include <stdio.h>
4
5  void displayBits( unsigned value ); /* prototipo */
6
7  int main()
8  {
9     unsigned x; /* variabile destinata a contenere l'input
                      dell'utente */
10
11    printf( "Enter an unsigned integer: " );
12    scanf( "%u", &x );
13
14    displayBits( x );
15
16    return 0; /* indica che il programma è terminato con successo */
17
18 } /* fine della funzione main */
19
20 /* visualizza i bit di un valore intero senza segno */
21 void displayBits( unsigned value )
```

Figura 10.7 Visualizzare in bit un intero senza segno (continua)

```
{
   unsigned c; /* contatore */

   /* dichiara displayMask ed esegue uno scorrimento a sinistra
      di 31 bit */
   unsigned displayMask = 1 << 31;

   printf( "%10u = ", value );

   /* itera scorrendo i bit */
   for ( c = 1; c <= 32; c++ ) {
      putchar(value & displayMask ? '1' : '0');
      value <<= 1; /* esegue uno scorrimento a sinistra di 1
                      su value */

      if ( c % 8 == 0 ) { /*manda in output uno spazio dopo 8 bit */
         putchar( ' ' );
      } /* fine del comando if */

   } /* fine del comando for */

   putchar( '\n' );
} /* fine della funzione displayBits */
```

```
Enter an unsigned integer: 65000
  65000 = 00000000 00000000 11111101 11101000
```

Figura 10.7 Visualizzare in bit un intero senza segno

La Figura 10.8 riepiloga i risultati che si ottengono dalla combinazione di due bit con l'operatore bitwise AND.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Figura 10.8 Risultati della combinazione di due bit con l'operatore bitwise AND &

Errore tipico 10.11

Usare l'operatore logico AND (&&) invece dell'operatore bitwise AND (&) e viceversa è un errore.

Utilizzare gli operatori bitwise AND, OR inclusivo, OR esclusivo e complemento

Il programma della Figura 10.9 dimostrerà l'uso degli operatori bitwise AND, OR inclusivo, OR esclusivo e complemento.Il programma userà la funzione `displayBits` (righe 53-74) per visualizzare i valori interi `unsigned`. L'output è mostrato nella Figura 10.10.

```
1  /* Fig. 10.9: fig10_09.c
2     Usare gli operatori biwise AND, OR inclusivo,
3     OR esclusivo e complemento */
4  #include <stdio.h>
5
6  void displayBits( unsigned value ); /* prototipo */
7
8  int main()
9  {
10    unsigned number1;
11    unsigned number2;
12    unsigned mask;
13    unsigned setBits;
14
15    /* illustra l'utilizzo dell'AND bitwise (&) */
16    number1 = 65535;
17    mask = 1;
18    printf( "The result of combining the following\n" );
19    displayBits( number1 );
20    displayBits( mask );
21    printf( "using the bitwise AND operator & is\n" );
22    displayBits( number1 & mask );
23
24    /* illustra l'utilizzo dell'OR inclusivo bitwise (|) */
25    number1 = 15;
26    setBits = 241;
27    printf( "\nThe result of combining the following\n" );
28    displayBits( number1 );
29    displayBits( setBits );
30    printf( "using the bitwise inclusive OR operator | is\n" );
31    displayBits( number1 | setBits );
32
33    /* illustra l'utilizzo dell'OR esclusivo bitwise (^) */
34    number1 = 139;
35    number2 = 199;
36    printf( "\nThe result of combining the following\n" );
37    displayBits( number1 );
38    displayBits( number2 );
39    printf( "using the bitwise exclusive OR operator ^ is\n" );
40    displayBits( number1 ^ number2 );
41
```

Figura 10.9 Usare gli operatori bitwise AND, OR inclusivo, OR esclusivo e complemento (continua)

```
42       /* illustra l'utilizzo del complemento bitwise (~) */
43       number1 = 21845;
44       printf( "\nThe one's complement of\n" );
45       displayBits( number1 );
46       printf( "is\n" );
47       displayBits( ~number1 );
48
49       return 0; /* indica che il programma è terminato con successo */
50    } /* fine della funzione main */
51
52    /* visualizza i bit di un valore intero senza segno */
53    void displayBits( unsigned value )
54    {
55       unsigned c; /* contatore */
56
57       /* dichiara displayMask ed esegue uno scorrimento a sinistra
            di 31 bit */
58       unsigned displayMask = 1 << 31;
59
60       printf( "%10u = ", value );
61
62       /* itera scorrendo i bit */
63       for ( c = 1; c <= 32; c++ ) {
64          putchar( value & displayMask ? '1' : '0' );
65          value <<= 1; /* esegue uno scorrimento a sinistra di 1
                             su value */
66
67          if (c % 8 == 0) { /*manda in output uno spazio dopo 8 bit */
68             putchar( ' ' );
69          } /* fine del comando if */
70
71       } /* fine del comando for */
72
73       putchar( '\n' );
74    } /* fine della funzione displayBits */
```

Figura 10.9 Usare gli operatori bitwise AND, OR inclusivo, OR esclusivo e complemento

Nella Figura 10.9, alla variabile intera `number1` sarà assegnato il valore `65535 (00000000 00000000 11111111 11111111)` nella riga 16 e alla variabile `mask` sarà assegnato il valore `1 (00000000 00000000 00000000 00000001)` nella riga 17. Nel momento in cui `number1` e `mask` saranno combinate usando l'operatore bitwise AND (`&`), il risultato dell'espressione `number1 & mask` (riga 22) sarà `00000000 00000000 00000000 00000001`. Tutti i bit nella variabile `number1`, eccettuato quello meno significativo, saranno "mascherati" (nascosti) "mettendoli in AND" con la variabile `mask`.

L'operatore bitwise OR inclusivo è usato per impostare a 1 determinati bit di un operando. Nella Figura 10.9, alla variabile `number1` sarà assegnato `15 (00000000 00000000 00000000 00001111)` nella riga 25 e alla variabile `setBits` sarà assegnato `241 (00000000 00000000 00000000 11110001)` nella riga 26. Nel momento in cui `number1` e `setBits`

```
The result of combining the following
  65535 = 00000000 00000000 11111111 11111111
      1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
      1 = 00000000 00000000 00000000 00000001

The result of combining the following
     15 = 00000000 00000000 00000000 00001111
    241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
    255 = 00000000 00000000 00000000 11111111

The result of combining the following
    139 = 00000000 00000000 00000000 10001011
    199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
     76 = 00000000 00000000 00000000 01001100

The one's complement of
  21845 = 00000000 00000000 01010101 01010101
is
  4294945450 = 11111111 11111111 10101010 10101010
```

Figura 10.10 L'output del programma della Figura 10.9

saranno combinate utilizzando l'operatore bitwise OR, il risultato dell'espressione `number1 | setBits` (riga 31) sarà 255 (`00000000 00000000 00000000 11111111`). La Figura 10.11 riepiloga i risultati che si ottengono dalla combinazione di due bit con l'operatore bitwise OR inclusivo.

Errore tipico 10.12

Usare l'operatore logico OR (`||`) invece dell'operatore bitwise OR (`|`) e viceversa è un errore.

L'operatore bitwise OR esclusivo (`^`) imposta a `1` ogni bit del risultato, qualora uno *solo* dei bit corrispondenti nei suoi due operandi sia 1. Nella Figura 10.9, alle variabili `number1` e `number2` sono stati assegnati i valori `139` (`00000000 00000000 00000000 10001011`) e `199` (`00000000 00000000 00000000 11000111`) nelle righe 34-35. Nel momento in cui queste variabili saranno combinate usando l'operatore OR esclusivo, il risultato dell'espressione `number1 ^ number2` (riga 40) sarà `00000000 00000000 00000000 01001100`. La Figura 10.12 riepiloga i risultati che si ottengono dalla combinazione di due bit con l'operatore bitwise OR esclusivo.

L'operatore bitwise di complemento (`~`) imposta a `0` tutti i bit del risultato corrispondenti a quelli che valgono `1` nel suo operando, mentre imposta a `1` tutti i bit del risultato corrispondenti a quelli che valgono `0` nel suo operando (operazione detta anche "prendere il *complemento a uno* del valore"). Nella Figura 10.9, alla variabile `number1` sarà assegnato il valore `21845` (`00000000 00000000 01010101 01010101`) nella riga 43. Nel momento in cui l'espressione `~number1` (riga 47) sarà valutata il risultato sarà (`11111111 11111111 10101010 10101010`).

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Figura 10.11 Risultati della combinazione di due bit con l'operatore bitwise OR inclusivo |

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Figura 10.12 Risultati della combinazione di due bit con l'operatore bitwise OR esclusivo ^

Utilizzare gli operatori bitwise di scorrimento a sinistra e di scorrimento a destra

Il programma della Figura 10.13 dimostrerà l'uso degli operatori di scorrimento a sinistra (<<) e a destra (>>). La funzione `displayBits` sarà usata per visualizzare dei valori interi `unsigned`.

L'operatore di scorrimento a sinistra (<<) fa scorrere in quella direzione i bit del suo operando di sinistra, per un numero di volte specificato da quello di destra. I bit svuotati a destra saranno sostituiti con degli `0`; gli `1` scivolati via da sinistra andranno persi. Nel programma della Figura 10.13, alla variabile `number1` sarà assegnato il valore `960 (00000000 00000000 00000011 11000000)` nella riga 9. Il risultato dello scorrimento a sinistra di 8 bit della variabile `number1` nell'espressione `number1 << 8` (riga 16) sarà `245760 (00000000 00000011 11000000 00000000)`.

L'operatore di scorrimento a destra (>>) fa scorrere in quella direzione i bit del suo operando di sinistra, per un numero di volte specificato da quello di destra. Eseguire uno scorrimento a destra su un intero `unsigned` provocherà la sostituzione dei bit rimasti vuoti a sinistra con degli `0`; gli `1` scivolati via da destra andranno persi. Nel programma della Figura 10.13, il risultato dello scorrimento a destra della variabile `number1` nell'espressione `number1 >> 8` (riga 23) sarà `3 (00000000 00000000 00000000 00000011)`.

```
1  /* Fig. 10.13: fig10_13.c
2     Usare gli operatori bitwise di scorrimento */
3  #include <stdio.h>
4
5  void displayBits( unsigned value ); /* prototipo */
6
```

Figura 10.13 Usare gli operatori bitwise di scorrimento (continua)

```
 7  int main()
 8  {
 9     unsigned number1 = 960; /* inizializza number1 */
10
11     /* illustra l'utilizzo dello scorrimento a sinistra bitwise */
12     printf( "\nThe result of left shifting\n" );
13     displayBits( number1 );
14     printf( "8 bit positions using the " );
15     printf( "left shift operator << is\n" );
16     displayBits( number1 << 8 );
17
18     /* illustra l'utilizzo dello scorrimento a destra bitwise */
19     printf( "\nThe result of right shifting\n" );
20     displayBits( number1 );
21     printf( "8 bit positions using the " );
22     printf( "right shift operator >> is\n" );
23     displayBits( number1 >> 8 );
24
25     return 0; /* indica che il programma è terminato con successo */
26  } /* fine della funzione main */
27
28  /* visualizza i bit di un valore intero senza segno */
29  void displayBits( unsigned value )
30  {
31     unsigned c; /* contatore */
32
33     /* dichiara displayMask ed esegue uno scorrimento a sinistra
          di 31 bit */
34     unsigned displayMask = 1 << 31;
35
36     printf( "%10u = ", value );
37
38     /* itera scorrendo i bit */
39     for (c = 1; c <= 32; c++) {
40        putchar( value & displayMask ? '1' : '0' );
41        value <<= 1; /* esegue uno scorrimento a sinistra di 1
                          su value */
42
43        if (c % 8 == 0) { /*manda in output uno spazio dopo 8 bit */
44           putchar( ' ' );
45        } /* fine del comando if */
46
47     } /* fine del comando for */
48
49     putchar( '\n' );
50  } /* fine della funzione displayBits */
```

Figura 10.13 Usare gli operatori bitwise di scorrimento (continua)

```
The result of left shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
  245760 = 00000000 00000011 11000000 00000000

The result of right shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000000 00000000 00000011
```

Figura 10.13 Usare gli operatori bitwise di scorrimento

Errore tipico 10.13

Il risultato dello scorrimento di un valore sarà indefinito, qualora l'operando di destra sia negativo o sia maggiore del numero di bit in cui è immagazzinato l'operando di sinistra.

Obiettivo portabilità 10.7

Il risultato di uno scorrimento a destra dipende dalla macchina. Lo scorrimento a destra di un intero con segno riempirà i bit svuotati con `0`*, su alcune macchine, e con* `1` *su altre.*

Operatori di assegnamento bitwise

Tutti gli operatori binari bitwise hanno un corrispondente operatore di assegnamento. Nella Figura 10.14, sono mostrati gli *operatori di assegnamento bitwise*; questi operatori sono usati in modo simile a quelli di assegnamento aritmetici introdotti nel Capitolo 3.

Operatori di assegnamento bitwise	
`&=`	Operatore di assegnamento bitwise AND.
`|=`	Operatore di assegnamento bitwise OR inclusivo.
`^=`	Operatore di assegnamento bitwise OR esclusivo.
`<<=`	Operatore di assegnamento scorrimento a sinistra.
`>>=`	Operatore di assegnamento scorrimento a destra.

Figura 10.14 Gli operatori di assegnamento bitwise

La Figura 10.15 mostra la priorità e l'associatività dei vari operatori introdotti fino a questo punto nel testo. I suddetti sono mostrati, dall'alto in basso, in ordine decrescente di precedenza.

Operatore	Associatività	Tipo
`() [] . ->`	da sinistra a destra	massima
`+ - ++ -- ! & * ~ sizeof (`*tipo*`)`	da destra a sinistra	unario
`* / %`	da sinistra a destra	moltiplicativo
`+ -`	da sinistra a destra	additivo
`<< >>`	da sinistra a destra	scorrimento
`< <= > >=`	da sinistra a destra	relazionale
`== !=`	da sinistra a destra	uguaglianza
`&`	da sinistra a destra	AND bitwise
`^`	da sinistra a destra	OR bitwise esclusivo
`|`	da sinistra a destra	OR bitwise inclusivo
`&&`	da sinistra a destra	AND logico
`||`	da sinistra a destra	OR logico
`?:`	da destra a sinistra	condizionale
`= += -= *= /= &= |= ^= <<= >>= %=`	da destra a sinistra	assegnamento
`,`	da sinistra a destra	virgola

Figura 10.15 Priorità e associatività degli operatori

10.10 I campi di bit

Il C consente ai programmatori di specificare in quanti bit dovrà essere immagazzinato un membro `unsigned` o `int` di una struttura o di un'unione. Questa caratteristica è detta *campo di bit*. I campi di bit consentono un migliore utilizzo della memoria, immagazzinando i dati nel minor numero di bit necessario. I membri di un campo di bit *devono* essere dichiarati come `int` o `unsigned`.

Obiettivo efficienza 10.3

I campi di bit aiutano a risparmiare la memoria.

Considerate la seguente definizione di struttura:

```
struct bitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

La definizione contiene tre campi di bit `unsigned` (`face`, `suit` e `color`) usati per rappresentare una delle 52 carte di un mazzo. Un campo di bit è dichiarato facendo seguire al *nome di un membro* `unsigned` o `int` il carattere due punti (`:`) e una costante intera, che rappresenta la *dimensione* del campo (vale a dire il numero di bit in cui sarà immagazzinato il membro).

La costante che rappresenta la dimensione dovrà essere un intero compreso tra 0 e il numero totale di bit usato per immagazzinare un `int` sul vostro sistema. I nostri esempi sono stati provati su un computer con interi di 4 byte (32 bit).

La definizione di struttura precedente indica che il membro `face` sarà immagazzinato in 4 bit, `suit` in 2, e `color` in 1. Il numero dei bit è basato sull'intervallo di valori desiderato per ognuno dei membri inclusi nella struttura. Il membro `face` immagazzinerà valori compresi tra `0` (Asso) e `12` (Re), e 4 bit sono in grado di memorizzare valori compresi tra `0` e `15`. Il membro `suit` immagazzinerà valori compresi tra `0` e `3` (`0` = Quadri, `1` = Cuori, `2` = Fiori, `3` = Picche), e 2 bit sono in grado di memorizzare valori compresi tra `0` e `3`. Il membro `color` infine immagazzinerà `0` (Rosso) o `1` (Nero), e 1 bit è appunto in grado di memorizzare `0` o `1`.

Il programma della Figura 10.16 (il cui output è mostrato nella Figura 10.17) creerà il vettore `deck` contenente 52 strutture di tipo `struct bitCard` nella riga 20. La funzione `fillDeck` (righe 30-41) inserirà le 52 carte nel vettore `deck` e `deal` (righe 45-58) le visualizzerà. Osservate che ai membri di tipo campo di bit delle strutture si accede esattamente come a ogni loro altro membro. Il membro `color` è stato incluso con l'intenzione di indicare il colore della carta, per un sistema che consenta la visualizzazione dei colori.

```
1  /* Fig. 10.16: fig10_16.c
2     Rappresentare le carte con i campi di bit in una struct */
3
4  #include <stdio.h>
5
6  /* dichiarazione della struttura bitCard con i campi di bit */
7  struct bitCard {
8     unsigned face : 4;   /* 4 bit: 0-15 */
9     unsigned suit : 2;   /* 2 bit: 0-3 */
10    unsigned color : 1;  /* 1 bit: 0-1 */
11 }; /* fine della struct bitCard */
12
13 typedef struct bitCard Card; /* nuovo nome di tipo per struct
                                   bitCard */
14
15 void fillDeck( Card * const wDeck ); /* prototipo */
16 void deal( Card * const wDeck );     /* prototipo */
17
18 int main()
19 {
20    Card deck[ 52 ]; /* crea un vettore di elementi di tipo Card */
21
22    fillDeck( deck );
23    deal( deck );
24
25    return 0; /* indica che il programma è terminato con successo */
26
27 } /* fine della funzione main */
28
```

Figura 10.16 Usare i campi di bit per immagazzinare un mazzo di carte (continua)

```
29  /* inizializza gli elementi di tipo Card */
30  void fillDeck( Card * const wDeck )
31  {
32     int i; /* contatore */
33
34     /* itera scorrendo gli elementi di wDeck */
35     for ( i = 0; i <= 51; i++ ) {
36        wDeck[ i ].face = i % 13;
37        wDeck[ i ].suit = i / 13;
38        wDeck[ i ].color = i / 26;
39     } /* fine del comando for */
40
41  } /* fine della funzione fillDeck */
42
43  /* La funzione deal visualizza le carte in un formato a due
       colonne; le carte da 0 a 25 indicizzate da  k1 (colonna 1);
44     le carte da 26 a 51 indicizzate da k2 (colonna 2) */
45  void deal( const Card * const wDeck )
46  {
47     int k1; /* indicizza 0-25 */
48     int k2; /* indicizza 26-51 */
49
50     /* itera scorrendo gli elementi di wDeck */
51     for ( k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ ) {
52        printf("Card:%3d  Suit:%2d  Color:%2d   ",
53              wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color);
54        printf("Card:%3d  Suit:%2d  Color:%2d\n",
55              wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color);
56     } /* fine del comando for */
57
58  } /* fine della funzione deal */
```

Figura 10.16 Usare i campi di bit per immagazzinare un mazzo di carte

```
Card:  0  Suit: 0  Color: 0   Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0   Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0   Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0   Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0   Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0   Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0   Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0   Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0   Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0   Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0   Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0   Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0   Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0   Card:  0  Suit: 3  Color: 1
```

Figura 10.17 L'output del programma nella Figura 10.16 (continua)

```
Card:  1  Suit: 1  Color: 0   Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0   Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0   Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0   Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0   Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0   Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0   Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0   Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0   Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0   Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0   Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0   Card: 12  Suit: 3  Color: 1
```

Figura 10.17 L'output del programma nella Figura 10.16

È possibile specificare un *campo di bit non denominato* da usare come un *riempitivo*. Per esempio, la definizione di struttura

```
struct example {
   unsigned a : 13;
   unsigned   : 19;
   unsigned b : 4;
};
```

usa come riempitivo un campo non denominato di 19 bit (in questi non si potrà immagazzinare niente). Il membro `b` (sul nostro computer con una word di 4 byte) sarà immagazzinato in un'altra unità di memoria.

Un *campo di bit non denominato con dimensione zero* è usato per allineare il campo di bit successivo su un nuovo confine di unità di memoria. Per esempio, la definizione di struttura

```
struct example {
   unsigned a : 13;
   unsigned   : 0;
   unsigned b : 4;
};
```

userà un campo non denominato di `0` bit, per saltare i bit rimanenti (tutti quelli che ci sono) dell'unità di memoria in cui sarà stata immagazzinata `a`, allineando `b` con il confine successivo dell'unità di memoria.

Obiettivo portabilità 10.8

I risultati delle manipolazioni dei campi di bit dipendono dalla macchina. Alcuni computer, per esempio, permettono che i campi di bit attraversino i confini di una parola, mentre altri non lo consentono.

Errore tipico 10.14

Tentare di accedere ai singoli bit di un campo di bit come se fossero elementi di un vettore è un errore di sintassi. I campi di bit non sono "vettori di bit".

Errore tipico 10.15

Tentare di rilevare l'indirizzo di un campo di bit (l'operatore & non può essere usato con i campi di bit, perché questi non hanno un indirizzo).

Obiettivo efficienza 10.4

Per quanto consentano di risparmiare dello spazio, i campi di bit potrebbero costringere il compilatore a generare un codice in linguaggio macchina più lento. Ciò accade perché, per accedere alle porzioni di un'unità di memoria indirizzabile, sono necessarie delle operazioni aggiuntive in linguaggio macchina. Questo è uno dei tanti esempi del genere di compromessi tra spazio e tempo che si presentano in informatica.

10.11 Le costanti di enumerazione

Il C fornisce un ultimo tipo di dato definibile dall'utente: l'*enumerazione*. Introdotta dalla parola chiave `enum`, un'enumerazione è un insieme di *costanti di enumerazione* intere rappresentate da identificatori. I valori di una `enum` incominciano da `0`, sempre che non sia stato diversamente specificato, e sono incrementati di `1`. Per esempio, l'enumerazione

```
enum months {
      JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

creerà un nuovo tipo, `enum months`, in cui gli identificatori saranno impostati con gli interi compresi tra `0` e `11`. Per numerare i mesi con i valori compresi tra 1 e 12, usate la seguente enumerazione:

```
enum months {
      JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
```

Dato che il primo valore della precedente enumerazione sarà impostato esplicitamente a `1`, i valori rimanenti saranno incrementati incominciando da `1` e producendo valori compresi tra 1 e 12. Gli identificatori inclusi in un'enumerazione devono essere univoci. Il valore di ogni costante di enumerazione può essere impostato esplicitamente all'interno della definizione, assegnando un valore all'identificatore. Uno stesso valore costante potrà essere assegnato a vari membri di un'enumerazione. Nel programma della Figura 10.18, la variabile di enumerazione `month` sarà usata in un comando `for` per visualizzare i mesi dell'anno inclusi nel vettore `monthName`. Notate che abbiamo assegnato a `monthName[0]` una stringa vuota `""`. Altri programmatori preferirebbero invece assegnare a `monthName[0]` un valore come `***ERROR***`, per indicare l'occorrenza di un errore logico.

Errore tipico 10.16

Assegnare un valore a una costante di enumerazione, dopo che sia stata definita, è un errore di sintassi.

Buona abitudine 10.5

Usate soltanto le lettere maiuscole nei nomi delle costanti di enumerazione. Ciò evidenzierà tali costanti all'interno di un programma e ricorderà al programmatore che le costanti di enumerazione non sono delle variabili.

```
1  /* Fig. 10.18: fig10_18.c
2     Usare il tipo enumerazione */
3  #include <stdio.h>
4
5  /* costanti di enumerazione che rappresentano i mesi dell'anno */
6  enum months {
7     JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
8
9  int main()
10 {
11    enum months month; /* può contenere uno qualunque dei 12 mesi */
12
13    /* inizializza il vettore di puntatori */
14    char *monthName[] = { "", "January", "February", "March",
15        "April", "May", "June", "July", "August", "September",
16                        "October", "November", "December"};
17
18    /* itera scorrendo i mesi */
19    for ( month = JAN; month <= DEC; month++ ) {
20       printf( "%2d%11s\n", month, monthName[month] );
21    } /* fine del comando for */
22
23    return 0; /* indica che il programma è terminato con successo */
24 } /* fine della funzione main */
```

```
 1   January
 2  February
 3     March
 4     April
 5       May
 6      June
 7      July
 8    August
 9 September
10   October
11  November
12  December
```

Figura 10.18 Usare un'enumerazione

Esercizi di autovalutazione

10.1 Riempite gli spazi in ognuna delle seguenti righe:

a) Una _____ è una collezione di variabili correlate sotto un unico nome.

b) Una _____ è una collezione di variabili sotto un unico nome in cui esse condividono lo stesso spazio di memoria.

c) I bit del risultato di un'espressione che utilizzi l'operatore _____ saranno impostati a 1, qualora quelli corrispondenti in ognuno degli operandi siano impostati a 1. Altrimenti, saranno impostati a zero.

d) Le variabili dichiarate in una definizione di struttura sono i suoi _____.

e) I bit nel risultato di un'espressione che utilizzi l'operatore _____ saranno impostati a 1, qualora almeno uno di quelli corrispondenti in uno o l'altro degli operandi sia impostato a 1. Altrimenti, saranno impostati a zero.
f) La parola chiave _____ introduce una dichiarazione di struttura.
g) La parola chiave _____ è usata per creare un sinonimo per un tipo di dato definito in precedenza.
h) I bit nel risultato di un'espressione che utilizzi l'operatore _____ saranno impostati a 1, qualora solo uno di quelli corrispondenti nei suoi due operandi sia impostato a 1. Altrimenti, saranno impostati a zero.
i) L'operatore bitwise AND `&` è usato spesso per _____ i bit, vale a dire per selezionarne alcuni, mentre si azzerano gli altri.
j) La parola chiave _____ è usata per introdurre una definizione di unione.
k) Il nome della struttura è detto anche structure _____.
l) Si accede a un membro di una struttura con l'operatore _____ o con l'operatore _____.
m) Gli operatori _____ e _____ sono usati per far scorrere rispettivamente a sinistra e a destra i bit di un certo valore.
n) Un'_____ è un insieme di interi rappresentato da identificatori.

10.2 Determinate se ognuna delle seguenti affermazioni sia *vera* o *falsa*. Qualora sia *falsa*, spiegatene il motivo:

a) Le strutture possono contenere variabili aventi soltanto un tipo di dato.
b) Due unioni possono essere confrontate (utilizzando `==`) per determinare se sono uguali.
c) Il tag name di una struttura è opzionale.
d) I membri di strutture differenti devono avere nomi univoci.
e) La parola chiave `typedef` è usata per definire nuovi tipi di dato.
f) Le strutture sono sempre passate per riferimento nelle chiamate di funzione.
g) Le strutture non possono essere confrontate utilizzando gli operatori `==` e `!=`.

10.3 Scrivete del codice che svolga ognuno dei seguenti compiti:

a) Definite una struttura chiamata `part` contenente la variabile `int partNumber` e il vettore di `char partName`, i cui valori possono essere lunghi 25 caratteri (includendo il carattere nullo di terminazione).
b) Definite `Part` come sinonimo del tipo `struct part`.
c) Usando `Part` dichiarate la variabile `a` e il vettore `b[10]` di tipo `struct part`, e la variabile `ptr` come puntatore a `struct part`.
d) Leggete dalla tastiera un numero e un nome di parte e immagazzinateli nei singoli membri della variabile `a`.
e) Assegnate all'elemento 3 del vettore `b` i valori contenuti nei membri della variabile `a`.
f) Assegnate alla variabile puntatore `ptr` l'indirizzo del vettore `b`.
g) Usando la variabile `ptr` e l'operatore puntatore a struttura per puntare ai membri, visualizzate i valori contenuti in quello dell'elemento 3 del vettore `b`.

10.4 Trovate l'errore in ognuno dei seguenti esercizi:

a) Supponete che `struct card` sia stata definita e che contenga due puntatori al tipo `char`, chiamati `face` e `suit`. Supponete inoltre che siano state dichiarate le variabili `c` di tipo `struct card` e `cPtr` come puntatore a `struct card`, e che alla variabile `cPtr` sia stato assegnato l'indirizzo di `c`.

```
printf( "%s\n", *cPtr->face );
```

b) Supponete che `struct card` sia già stata definita e che contenga due puntatori al tipo `char`, chiamati `face` e `suit`. Supponete inoltre che sia stato dichiarato il vettore `hearts[13]`

di tipo `struct card`. L'istruzione seguente dovrebbe visualizzare il membro `face` dell'elemento 10 del vettore.

```
printf( "%s\n", hearts.face );
```

c)
```
union values {
   char w;
   float x;
   double y;
};

union values v = { 1.27 };
```

d)
```
struct person {
   char lastName[ 15 ];
   char firstName[ 15 ];
   int age;
}
```

e) Assumete che `struct person` sia stata dichiarata come nella parte (d), ma con l'appropriata correzione.

```
person d;
```

f) Assumete che siano state dichiarate le variabili p di tipo `struct person` e c di tipo `struct card`.

```
p = c;
```

Risposte agli esercizi di autovalutazione

10.1 a) struttura; b) unione; c) AND bitwise (`&`); d) membri; e) OR inclusivo bitwise (`|`); f) `struct`; g) `typedef`; h) OR esclusivo bitwise (`^`); i) mascherare; j) `union`; k) tag; l) membro di struttura, puntatore a struttura; m) operatore di scorrimento a sinistra (`<<`), operatore di scorrimento a destra (`>>`); n) enumerazione.

10.2 a) Falso. Una struttura può contenere variabili aventi molti tipi di dato.
b) Falso. Le unioni non possono essere confrontate, poiché ci potrebbero essere dei byte di dati non definiti contenenti valori diversi in variabili di unioni che altrimenti risulterebbero uguali.
c) Vero.
d) Falso. I membri di strutture distinte possono avere lo stesso nome, mentre quelli della medesima struttura devono avere nomi univoci.
e) Falso. La parola chiave `typedef` è usata per definire nuovi nomi (sinonimi) per tipi di dato definiti in precedenza.
f) Falso. Le strutture sono sempre passate per valore nelle chiamate di funzione.
g) Vero, a causa dei problemi di allineamento.

10.3 a)
```
struct part {
   int partNumber;
   char partName[ 25 ];
};
```

b)
```
typedef struct part Part;
```

c)
```
Part a, b[ 10 ], *ptr;
```

d)
```
scanf( "%d%s", &a.partNumber, &a.partName );
```

e)
```
b[ 3 ] = a;
```

f)
```
ptr = b;
```

g)
```
printf( "%d %s\n", (ptr + 3)->partNumber, (ptr + 3)->partName );
```

10.4 a) Le parentesi che dovrebbero racchiudere `*cPtr` sono state omesse, causando un ordine scorretto di valutazione dell'espressione. Quest'ultima dovrebbe essere

```
( *cPtr )->face
```

b) L'indice del vettore è stato omesso. L'espressione dovrebbe essere

```
hearts[ 10 ].face.
```

c) Un'unione può essere inizializzata soltanto con un valore dello stesso tipo del suo primo membro.

d) È richiesto un punto e virgola per terminare una definizione di struttura.

e) La parola chiave `struct` è stata omessa dalla dichiarazione della variabile. Quest'ultima dovrebbe essere

```
struct person d;
```

f) Le variabili di tipi di struttura differenti non possono essere assegnate l'una all'altra.

Esercizi

10.5 Fornite la definizione per ognuna delle seguenti strutture e unioni:

a) Una struttura `inventory` che contenga il vettore di caratteri `partName[ 30 ]`, l'intero `partNumber`, il numero in virgola mobile `price` e gli interi `stock` e `reorder`.

b) Un'unione `data` che contenga `char c`, `short s`, `long b`, `float f` e `double d`.

c) Una struttura chiamata `address` che contenga i vettori di caratteri `streetAddress[ 25 ]`, `city[ 20 ]`, `state[ 3 ]` e `zipCode[ 6 ]`.

d) Una struttura `student` che contenga i vettori di caratteri `firstName[ 15 ]` e `lastName[ 15 ]`, e la variabile `homeAddress` del tipo `struct address` definito nella parte (c).

e) La struttura `test` che contenga 16 campi di bit di lunghezza 1. I nomi dei campi di bit sono le lettere comprese tra `a` e `p`.

10.5 Date le seguenti definizioni di strutture e dichiarazioni di variabili,

```
struct customer {
   char lastName[ 15 ];
   char firstName[ 15 ];
   int customerNumber;

   struct {
      char phoneNumber[ 11 ];
      char address[ 50 ];
      char city[ 15 ];
      char state[ 3 ];
      char zipCode[ 6 ];
   } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

scrivete un'espressione separata che possa essere usata per accedere ai membri della struttura in ognuna delle seguenti parti.

a) Membro `lastName` della struttura `customerRecord`.

b) Membro `lastName` della struttura puntata da `customerPtr`.

c) Membro `firstName` della struttura `customerRecord`.

d) Membro `firstName` della struttura puntata da `customerPtr`.

e) Membro `customerNumber` della struttura `customerRecord`.
f) Membro `customerNumber` della struttura puntata da `customerPtr`.
g) Membro `phoneNumber` del membro `personal` della struttura `customerRecord`.
h) Membro `phoneNumber` del membro `personal` della struttura puntata da `customerPtr`.
i) Membro `address` del membro `personal` della struttura `customerRecord`.
j) Membro `address` del membro `personal` della struttura puntata da `customerPtr`.
k) Membro `city` del membro `personal` della struttura `customerRecord`.
l) Membro `city` del membro `personal` della struttura puntata da `customerPtr`.
m) Membro `state` del membro `personal` della struttura `customerRecord`.
n) Membro `state` del membro `personal` della struttura puntata da `customerPtr`.
o) Membro `zipCode` del membro `personal` della struttura `customerRecord`.
p) Membro `zipCode` del membro `personal` della struttura puntata da `customerPtr`.

10.7 Modificate il programma della Figura 10.16, in modo che mescoli le carte mediante l'uso di un mescolatore ad alta efficienza (come mostrato nella Figura 10.3). Visualizzate il mazzo risultante nel formato a due colonne mostrato nella Figura 10.4. Fate precedere ogni carta dal suo colore.

10.8 Create l'unione `integer` con i membri `char c`, `short s`, `int i` e `long b`. Scrivete un programma che accetti in input valori di tipo `char`, `short`, `int` e `long` e li immagazzini nelle variabili di un'unione di tipo `union integer`. Ogni variabile dell'unione dovrà essere visualizzata come un `char`, uno `short`, un `int` e un `long`. I valori saranno sempre stampati correttamente?

10.9 Create l'unione `floatingPoint` con i membri `float f`, `double d` e `long double x`. Scrivete un programma che accetti in input valori di tipo `float`, `double` e `long double`, e li immagazzini nelle variabili di un'unione di tipo `union floatingPoint`. Ogni variabile dell'unione dovrà essere visualizzata come un `float`, un `double` e un `long double`. I valori saranno sempre stampati correttamente?

10.10 Scrivete un programma che faccia scorrere a destra di 4 bit una variabile intera. Il programma dovrà visualizzare l'intero in bit, prima e dopo l'operazione di scorrimento. Il vostro sistema metterà degli 0 o degli 1 nei bit svuotati?

10.11 Modificate il programma della Figura 10.7, in modo che possa funzionare con interi di 2 byte, qualora il vostro computer usi interi di 2 byte.

10.12 Far scorrere a sinistra di 1 bit un intero `unsigned` è equivalente a moltiplicare il valore per 2. Scrivete la funzione `power2`, che accetti i due argomenti interi `number` e `pow` e calcoli

`number * 2`$^{\texttt{pow}}$

Usate l'operatore di scorrimento per calcolare il risultato. Il programma dovrà visualizzare i valori come interi e come bit.

10.13 L'operatore di scorrimento a sinistra può essere usato per comprimere i valori di due caratteri, in una variabile intera di tipo `unsigned`. Scrivete un programma che accetti in input dalla tastiera due caratteri e li passi alla funzione `packCharacters`. Per comprimere due caratteri in una variabile intera `unsigned`, assegnatele il primo carattere, fatela scorrere a sinistra per 8 posizioni e combinatela con il secondo carattere, usando l'operatore bitwise OR inclusivo. Il programma dovrà visualizzare i caratteri nel loro formato a bit, prima e dopo essere stati compressi nell'intero `unsigned`, per dimostrare che saranno stati compressi correttamente nella variabile `unsigned`.

10.14 Usando gli operatori di scorrimento a destra, AND bitwise e una maschera, scrivete la funzione `unpackCharacters`, che prenda l'intero `unsigned` dell'Esercizio 10.13 e lo decomprima in due caratteri. Per decomprimere una coppia di caratteri da un intero `unsigned`, combinatelo con la maschera `65280 (00000000 00000000 11111111 00000000)` e fate scorrere a destra per otto bit il risultato. Assegnate il valore risultante a una variabile `char`. In seguito combinate l'intero `unsigned` con la maschera `255 (00000000 00000000 00000000 11111111)`. Assegnate il risultato a un'altra variabile

char. Il programma dovrà visualizzare in bit l'intero **unsigned**, prima della decompressione, e quindi dovrà visualizzare in bit anche i caratteri, per confermare che la loro decompressione sia stata eseguita correttamente.

10.15 Riscrivete il programma dell'Esercizio 10.13 in modo che comprima 4 caratteri, qualora il vostro sistema usi interi di 4 byte.

10.16 Riscrivete la funzione `unpackCharacters` dell'Esercizio 10.14 in modo che decomprima 4 caratteri, qualora il vostro sistema usi interi di 4 byte. Create le maschere necessarie per decomprimere i 4 caratteri, facendo scorrere a sinistra per 8 bit il valore 255 contenuto nella variabile maschera, per 0, 1, 2 o 3 volte (secondo il byte che state decomprimendo).

10.17 Scrivete un programma che inverta l'ordine dei bit di un valore intero senza segno. Il programma dovrà prendere in input dall'utente il valore e richiamare la funzione `reverseBits` per visualizzare i bit in ordine inverso. Visualizzate il valore in bit, prima e dopo che questi siano stati invertiti, per confermare che siano stati invertiti correttamente.

10.18 Modificate la funzione `displayBits` della Figura 10.7 in modo che sia portabile tra sistemi che usano interi di due o quattro byte. [*Suggerimento*: usate l'operatore `sizeof` per determinare la dimensione di un intero su una macchina particolare.]

10.19 Il seguente programma usa la funzione `multiple`, per determinare se l'intero immesso dalla tastiera è un multiplo di qualche intero `X`. Esaminate la funzione `multiple` e, quindi, determinate il valore di `X`.

```
1  /* ex10_19.c */
2     Questo programma determina se un valore è un multiplo di X */
3  #include <stdio.h>
4
5  int multiple( int num ); /*prototipo */
6
7  int main()
8  {
9     int y; /* y conterrà l'intero specificato dall'utente */
10
11    printf( "Enter an integer between 1 and 32000: " );
12    scanf("%d", &y);
13
14    /* se y è un multiplo di X */
15    if ( multiple( y ) ) {
16       printf( "%d is a multiple of X\n", y );
17    } /* fine del ramo if */
18    else {
19       printf( "%d is not a multiple of X\n", y );
20    } /* fine del ramo else */
21
22    return 0; /* indica che il programma è terminato con successo */
23 } /* fine della funzione main */
24
25 /* determina se num è un multiplo di X */
26 int multiple( int num )
27 {
28    int i; /* contatore */
```

```
   int mask = 1; /* inizializza mask */
   int mult = 1; /* inizializza mult */

   for ( i = 1; i <= 10; i++, mask <<= 1 ) {

      if ( ( num & mask ) != 0 ) {
         mult = 0;
         break;
      } /* fine del comando if */

   } /* fine del comando for */

   return mult;
} /* fine della funzione multiple */
```

10.20 Che cosa farà il programma seguente?

```
/* ex10_20.c */
#include <stdio.h>

int mystery( unsigned bits ); /* prototipo */

int main()
{
   unsigned x;

   printf( "Enter an integer: " );
 scanf( "%u", &x );

   printf( "The result is %d\n", mystery( x ) );

   return 0; /* indica che il programma è terminato con successo */
} /* fine della funzione main */

/* Che cosa farà questa funzione? */
int mystery( unsigned bits )
{
   unsigned i; /* contatore */
   unsigned mask = 1 << 31; /* inizializza mask */
   unsigned total = 0; /* inizializza total */

   for ( i = 1; i <= 16; i++, bits <<= 1 ) {

      if ( ( bits & mask ) == mask ) {
         ++total;
      } /* fine del comando if */

   } /* fine del comando for */

    return total % 2 == 0 ? 1 : 0;
} /* fine della funzione mystery */
```

Capitolo 11

L'elaborazione dei file in C

Obiettivi

- Essere in grado di creare, leggere, scrivere e modificare i file.
- Familiarizzare con l'elaborazione di file ad accesso sequenziale.
- Familiarizzare con l'elaborazione di file ad accesso casuale.

11.1 Introduzione

La memorizzazione dei dati nelle variabili e nei vettori è temporanea; tutti quei dati andranno persi, quando un programma terminerà la propria esecuzione. I *file* sono usati proprio per la conservazione permanente di grandi quantità di dati. I computer immagazzinano i file su dispositivi di memoria secondaria, specialmente su quelli a dischi. In questo capitolo, spiegheremo in che modo i file di dati siano creati, modificati ed elaborati dai programmi C. Prenderemo in considerazione i file ad accesso sequenziale e casuale.

11.2 La gerarchia dei dati

In ultima analisi, tutte le unità di informazione elaborate da un computer sono ridotte a combinazioni di zeri e di uno. Questo perché è semplice ed economico costruire dispositivi elettronici che possano assumere due stati stabili, dei quali l'uno rappresenti uno `0` e l'altro un `1`. È straordinario che le impressionanti funzioni eseguite dai computer comportino soltanto le più elementari manipolazioni di `0` e di `1`.

La più piccola unità di informazione in un computer può assumere il valore `0` o `1`. Tale unità di informazione è detta *bit* (abbreviazione per "*binary digit*" o "*cifra binaria*": un numero che può assumere uno tra due possibili valori). I circuiti dei computer eseguono diverse semplici manipolazioni del bit, come determinarne o impostarne il valore e invertire un bit (da `1` a `0` o da `0` a `1`).

Per i programmatori, è scomodo lavorare con i dati nella loro forma di basso livello fatta di bit. I programmatori preferiscono invece lavorare con dati in *cifre decimali* (ovverosia 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9), *lettere* (ovverosia A-Z e a-z) e *simboli speciali* (come $, @, %, &, *, (,), -, +, ", :, ?, /, e altri). I numeri, le lettere e i simboli speciali sono detti genericamente *caratteri*. L'insieme di tutti i caratteri che possono essere usati per scrivere i programmi e rappresentare le unità di informazione su un particolare computer è detto l'*insieme di caratteri* di quel computer. Dato che i computer possono elaborare soltanto i valori `1` e `0`, nell'insieme di caratteri del computer, ognuno di essi è rappresentato con un modello di `1` e `0` (detto *byte*). Oggigiorno, i byte sono composti per lo più da otto bit. I programmatori creeranno i programmi e le

unità di informazione usando i caratteri e, successivamente, questi saranno gestiti ed elaborati dai computer come modelli di bit.

I *campi* sono composti da caratteri, così come questi ultimi sono composti da bit. Un campo è un gruppo di caratteri che trasmette un'informazione. Per esempio, un campo consistente esclusivamente di lettere maiuscole e minuscole potrebbe essere usato per rappresentare il nome di una persona.

Le unità di informazione elaborate dai computer formano una *gerarchia di dati* in cui le unità di informazione diventano più grandi e strutturalmente più complesse, man mano che si procede dai bit ai caratteri (i byte), ai campi e così via.

Un *record* (ovverosia una `struct` in C) si compone di diversi campi. In un sistema di gestione degli stipendi, per esempio, il record di un particolare impiegato potrebbe consistere dei seguenti campi:

1. Numero di previdenza sociale (campo alfanumerico)
2. Nome (campo alfabetico)
3. Indirizzo (campo alfanumerico)
4. Importo orario del salario (campo numerico)
5. Numero di esenzioni rivendicate (campo numerico)
6. Progressivo dello stipendio nell'anno (campo numerico)
7. Ammontare delle tasse federali trattenute ecc. (campo numerico)

In altre parole, un record è un gruppo di campi correlati. Nell'esempio precedente, ognuno dei campi appartiene allo stesso impiegato. Naturalmente, una particolare azienda potrebbe avere molti impiegati e avrebbe un record di stipendio per ognuno di loro. Un *file* è un gruppo di record correlati. Il file degli stipendi di un'azienda contiene normalmente un record per ogni impiegato. Di conseguenza, il file degli stipendi di una piccola azienda potrebbe contenere soltanto 22 record, laddove lo stesso file in una grande azienda potrebbe contenerne 100.000. Per un'organizzazione, non è insolito avere centinaia o anche migliaia di file contenenti molti miliardi o anche migliaia di miliardi di caratteri di informazione. La Figura 11.1 mostra la gerarchia dei dati.

Per facilitare il recupero di record specifici da un file, almeno uno dei suoi campi è scelto come *chiave del record*. Questa lo identifica come appartenente a una particolare persona o entità. Per esempio, nel record degli stipendi descritto in questa sezione, normalmente sarà scelto come chiave del record il numero di previdenza sociale.

Ci sono molti modi di organizzare i record in un file. Il tipo di organizzazione più comune è il *file sequenziale*, in cui i record sono tipicamente immagazzinati in ordine in base al campo chiave del record. Il primo record di impiegato nel file contiene il numero di previdenza sociale più basso, mentre quelli successivi contengono numeri di previdenza sociale progressivamente più alti.

La maggior parte delle imprese archivia i dati in una moltitudine di file differenti. Per esempio, le aziende potrebbero avere dei file per gli stipendi, per i conti di credito (l'elenco del denaro dovuto dai clienti), per i conti di debito (l'elenco del denaro dovuto ai fornitori), per le scorte di magazzino (l'elenco dei fatti relativi agli articoli trattati nell'impresa) e molti altri tipi di file. A volte un gruppo di file correlati è detto *database*. Una collezione di programmi progettati per creare e gestire i database è detta *sistema per la gestione del database* (DBMS, Database Management System).

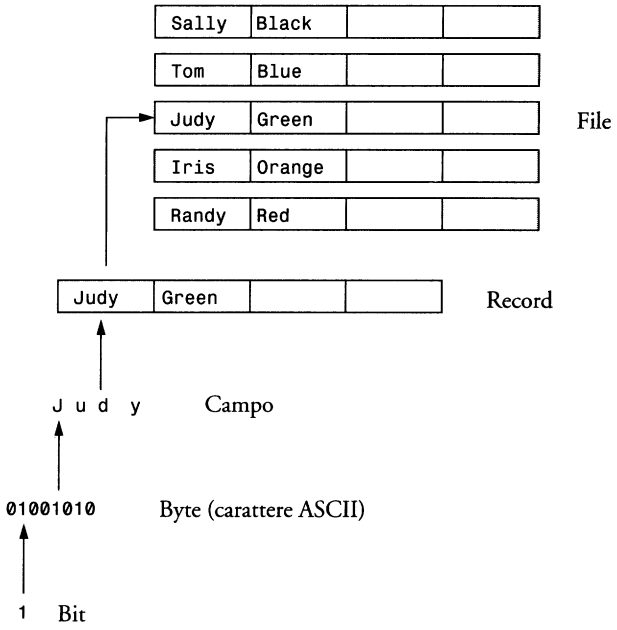


Figura 11.1 La gerarchia dei dati

11.3 I file e gli stream

Il C vede i file semplicemente come un flusso (stream) sequenziale di byte (Figura 11.2). Ogni file termina con un *marcatore di end-of-file* (fine del file), oppure a uno specifico numero di byte registrato in una struttura di dati amministrativi gestita dal sistema. Nel momento in cui un file sarà stato *aperto*, vi sarà associato uno stream. Tre file e i loro rispettivi stream, lo *standard input*, lo *standard output* e lo *standard error*, saranno aperti automaticamente, quando comincerà l'esecuzione di un programma. Gli stream forniscono un canale di comunicazione tra i file e i programmi. Per esempio, lo stream dello standard input consente a un programma di leggere i dati dalla tastiera, mentre quello dello standard output fa sì che un programma possa visualizzare i dati sullo schermo. L'apertura di un file restituisce un puntatore a una struttura `FILE` (definita in `<stdio.h>`) che contiene le informazioni utilizzate per elaborare il file. Tale struttura include un *descrittore di file* (*file descriptor*), ovverosia l'indice di un vettore del sistema operativo chiamata *tabella dei file aperti* (*open file table*). Ogni elemento del vettore contiene un *blocco di controllo del file* (*file control block* o *FCB*) che il sistema operativo utilizza per amministrare un particolare file. Lo standard input, lo standard output e lo standard error sono gestiti utilizzando rispettivamente i puntatori di file `stdin`, `stdout` e `stderr`.

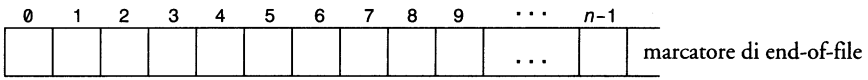


Figura 11.2 In che modo il C vede un file di *n* byte

La libreria standard fornisce molte funzioni per leggere e scrivere i dati dei file. La funzione `fgetc`, come `getchar`, legge un carattere da un file. La funzione `fgetc` riceve come argomento un puntatore a una struttura `FILE` per il file dal quale il carattere sarà letto. La chiamata `fgetc( stdin )` legge un carattere dallo `stdin`, ovverosia dallo standard input. Tale chiamata è quindi equivalente a `getchar()`. La funzione `fputc`, come `putchar`, scrive un carattere in un file. La funzione `fputc` riceve come argomento il carattere da scrivere e un puntatore al file in cui il carattere sarà scritto. La chiamata di funzione `fputc( 'a', stdout )` scrive il carattere `'a'` nello `stdout`, ovverosia nello standard output. Tale chiamata è quindi equivalente a `putchar( 'a' )`.

Tra le funzioni usate per leggere i dati dallo standard input e scriverli nello standard output, ce ne sono molte altre che hanno delle funzioni corrispondenti, chiamate in modo simile, tra quelle per l'elaborazione dei file. Le funzioni `fgets` e `fputs`, per esempio, potrebbero essere usate rispettivamente per leggere o per scrivere una riga di un file. Le loro controparti per leggere dallo standard input e per scrivere nello standard output, `gets` e `puts`, sono state discusse nel Capitolo 8. Nelle prossime sezioni, introdurremo le funzioni per l'elaborazione di file equivalenti a `scanf` e `printf`: `fscanf` e `fprintf`. In seguito, sempre in questo capitolo, discuteremo delle funzioni `fread` e `fwrite`.

11.4 Creare un file ad accesso sequenziale

Il C non impone una struttura ai file. Di conseguenza, nozioni come il record di un file non appartengono al linguaggio C. Il programmatore dovrà quindi fornire una struttura di file, per soddisfare le necessità di una particolare applicazione. Nell'esempio successivo, vedremo in che modo il programmatore potrà imporre una struttura di record a un file.

Il programma della Figura 11.3 creerà un semplice file ad accesso sequenziale, che potrebbe essere usato in un sistema di contabilità del credito, per aiutare a tenere il computo delle somme dovute dai clienti debitori di un'azienda. Per ogni cliente, il programma otterrà il suo numero di conto, il nome e il saldo (ovverosia l'ammontare dovuto all'azienda per i beni e i servizi ricevuti in passato dal cliente). I dati ottenuti per ogni cliente costituiranno il "record" dedicato a ognuno di loro. In questa applicazione, il numero di conto sarà usato come chiave del record: il file sarà dunque creato e mantenuto in ordine rispetto a quel dato. Questo programma presumerà che l'utente immetterà i record seguendo l'ordine dei numeri di conto. In un sistema completo di contabilità del credito, dovrebbe essere fornita la possibilità di eseguire degli ordinamenti, in modo che l'utente possa immettere i record in qualsiasi ordine. In seguito, i record dovrebbero essere ordinati e scritti nel file.

```
1  /* Fig. 11.3: fig11_03.c
2     Creare un file sequenziale */
3  #include <stdio.h>
4
5  int main()
6  {
7     int account; /* numero del conto */
8     char name[ 30 ]; /* nome dell'intestatario del conto */
9     double balance; /* bilancio del conto */
```

Figura 11.3 Creare un file sequenziale (continua)

```
10
11      FILE *cfPtr;    /* cfPtr = puntatore al file clients.dat */
12
13      /* fopen apre il file. Se non riesce a creare il file,
           provoca l'uscita dal programma */
14      if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
15         printf( "File could not be opened\n" );
16      } /* fine del ramo if */
17      else {
18         printf( "Enter the account, name, and balance.\n" );
19         printf( "Enter EOF to end input.\n" );
20         printf( "? " );
21         scanf( "%d%s%f", &account, name, &balance );
22
23         /* scrive il conto, il nome e il bilancio nel file,
              utilizzando fprintf */
24         while ( !feof( stdin ) ) {
25            fprintf( cfPtr, "%d %s %.2f\n", account, name, balance);
26            printf( "? " );
27            scanf( "%d%s%f", &account, name, &balance );
28         } /* fine del comando while */
29
30         fclose(cfPtr); /* fclose chiude il file */
31      } /* fine del ramo else */
32
33      return 0; /* indica che il programma è terminato con successo */
34
35   } /* fine della funzione main */
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Figura 11.3 Creare un file sequenziale

Esaminiamo ora questo programma. La riga 11

```
FILE *cfPtr;    /* cfPtr = puntatore al file clients.dat */
```

stabilisce che `cfPtr` è un puntatore a una struttura `FILE`. Un programma C gestisce ogni file con una struttura `FILE` distinta. Per usare i file, il programmatore non ha bisogno di conoscere i dettagli della struttura `FILE`. Vedremo presto, in modo preciso, come la struttura `FILE` conduca indirettamente al blocco di controllo del file (FCB) gestito dal sistema operativo.

Ogni file aperto deve avere un puntatore di tipo `FILE` dichiarato separatamente che sarà usato per puntare allo stesso. La riga 14

```
if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL ) {
```

assegnerà un nome, `"clients.dat"`, al file che dovrà essere usato dal programma e stabilirà una "linea di comunicazione" con esso. A `cfPtr` sarà assegnato un puntatore alla struttura `FILE` del file aperto con `fopen`. La funzione `fopen` riceve due argomenti: un nome e un *modo di apertura del file*. Il modo di apertura `"w"` indica che il file dovrà essere aperto in *scrittura*. Nel caso in cui il file non esista e venga aperto in scrittura, `fopen` lo creerà. Nel caso in cui un file già esistente fosse aperto in scrittura, i suoi contenuti saranno eliminati senza alcun avviso. All'interno del programma, il comando `if` sarà usato per determinare se il puntatore `cfPtr` sia `NULL` (ovverosia, se il file non sia stato aperto). Nel caso in cui sia `NULL`, il programma visualizzerà un messaggio di errore e terminerà la propria esecuzione. Altrimenti, il programma elaborerà l'input e lo scriverà nel file.

Errore tipico 11.1

Aprire un file già esistente in scrittura `("w")` *quando, di fatto, l'utente desideri preservare il file eliminerà i contenuti dello stesso senza alcun avviso.*

Errore tipico 11.2

Dimenticare di aprire un file, prima di tentare di farvi riferimento all'interno di un programma è un errore logico.

Il programma richiederà all'utente di immettere per ogni record i vari campi, o un end-of-file qualora l'immissione dei dati sia stata completata. La Figura 11.4 elenca le combinazioni di tasti utilizzate per immettere un end-of-file su diversi sistemi di computer.

La riga 24

```
while ( !feof( stdin ) ) {
```

userà la funzione `feof` per determinare se sia stato impostato l'*indicatore di end-of-file* per il file al quale punta `stdin`. L'indicatore di end-of-file informerà il programma che non ci saranno più dati da elaborare. Nel programma della Figura 11.3, l'indicatore di end-of-file sarà impostato per lo standard input, quando l'utente immetterà l'apposita combinazione di tasti. L'argomento della funzione `feof` è un puntatore al file del quale si desidera esaminare l'indicatore di end-of-file (`stdin` in questo caso). La funzione restituisce un valore diverso da zero (vero) quando l'indicatore di end-of-file è stato impostato; altrimenti, la funzione restituisce uno zero. Il comando `while`, che in questo programma include la chiamata a `feof`, continuerà la propria esecuzione fin quando non sarà stato impostato l'indicatore di end-of-file.

Sistema di computer	Combinazione di tasti
UNIX	*<return> <ctrl> d*
Windows	*<ctrl> z*
Macintosh	*<ctrl> d*

Figura 11.4 Le combinazioni di tasti per l'end-of-file su alcuni dei sistemi di computer più diffusi

La riga 25

```
fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
```

scriverà i dati nel file `clients.dat`. Questi potranno essere recuperati in seguito da un programma progettato per leggere il file (consultate la Sezione 11.5). La funzione `fprintf` è equivalente a `printf`, eccetto che `fprintf` riceve come argomento anche un puntatore al file in cui i dati saranno scritti.

Errore tipico 11.3

Usare il puntatore sbagliato per fare riferimento a un file è un errore logico.

Collaudo e messa a punto 11.1

Assicurarsi che, all'interno di un programma, le chiamate a funzioni per l'elaborazione dei file contengano i puntatori corretti.

Una volta che l'utente avrà immesso l'end-of-file, il programma chiuderà il file `clients.dat` con `fclose` e terminerà la propria esecuzione. Anche la funzione `fclose` riceve come argomento un puntatore a un file, invece del suo nome. Normalmente, nel caso in cui la funzione `fclose` non sia stata richiamata esplicitamente, il sistema operativo baderà a chiudere il file quando l'esecuzione del programma sarà terminata. Questo è un esempio delle attività di "ordinaria amministrazione" del sistema operativo.

Buona abitudine 11.1

Chiudete esplicitamente ogni file, non appena siete sicuri che il programma non vi farà più riferimento.

Obiettivo efficienza 11.1

La chiusura di un file potrebbe liberare risorse per le quali potrebbero essere in attesa altri utenti o programmi.

Nell'esempio di esecuzione per il programma della Figura 11.3, l'utente ha digitato informazioni per cinque conti e, alla fine, ha immesso l'end-of-file per segnalare che il caricamento dei dati è stato completato. L'esempio di esecuzione non mostra però come i record di dati appaiono realmente all'interno del file. Per verificare che questo sia stato creato correttamente, nella prossima sezione presenteremo un programma che leggerà il file e ne visualizzerà il contenuto.

La Figura 11.5 illustra le relazioni tra i puntatori a `FILE`, le strutture `FILE` e i FCB in memoria. Nel momento in cui il file `"clients.dat"` sarà stato aperto, il suo FCB sarà ricopiato in memoria. La figura mostra la connessione tra il puntatore restituito dalla `fopen` e il FCB usato dal sistema operativo per gestire il file.

Un programma può elaborare uno, nessuno o molti file. Ognuno di quelli usati in un programma dovrà avere un nome univoco e avrà un puntatore di file distinto che sarà restituito da `fopen`. Tutte le successive funzioni per l'elaborazione di un file, una volta che sarà stato aperto, dovranno farvi riferimento con il puntatore appropriato. I file potranno essere aperti in uno dei tanti modi disponibili (Figura 11.6). Per creare un file, o per eliminarne il contenuto prima di scrivere i dati, apritelo in scrittura (`"w"`). Per leggere un file già esistente,

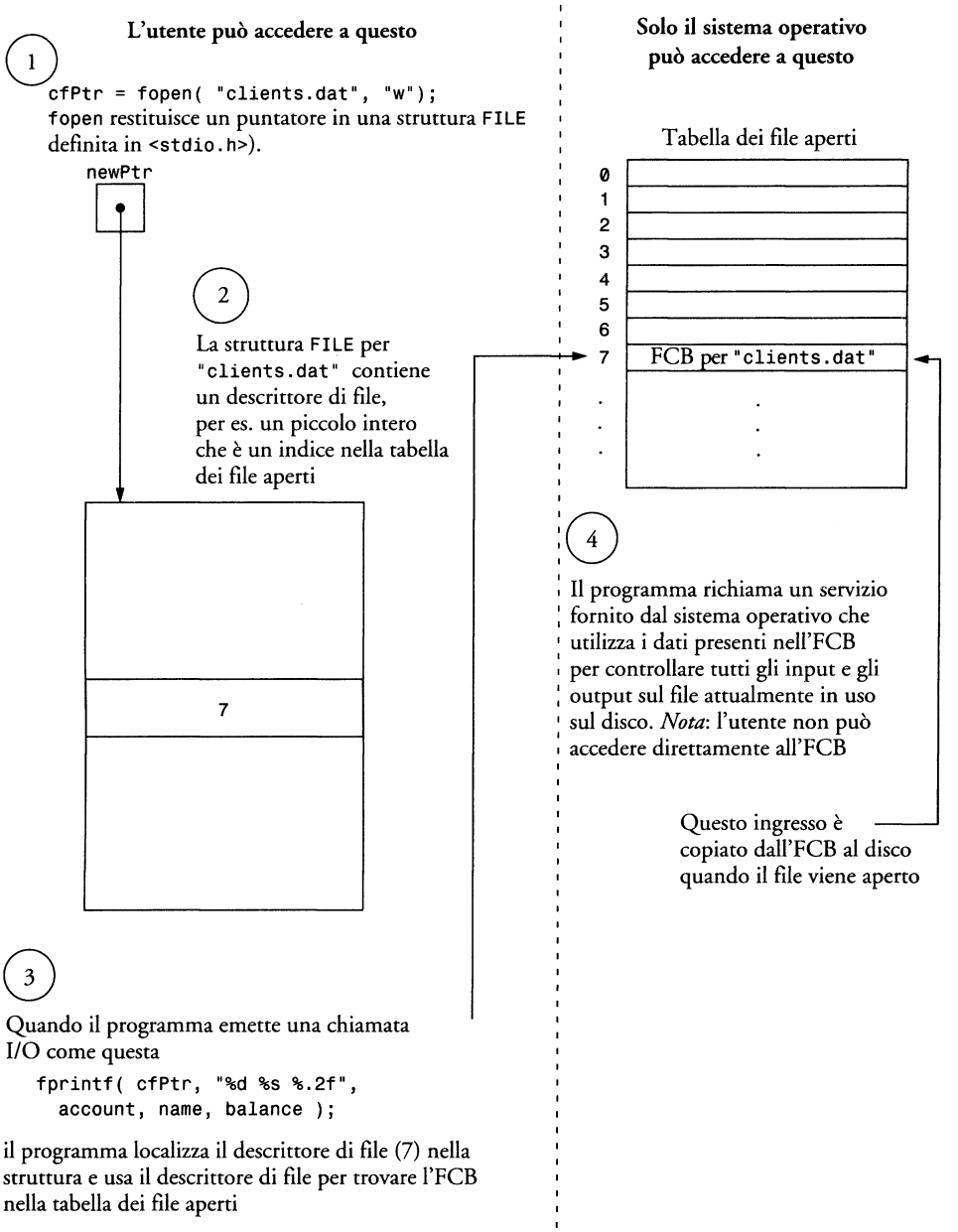


Figura 11.5 La relazione tra puntatori a `FILE`, le strutture `FILE` e i FCB

apritelo in lettura (`"r"`). Per aggiungere dei record alla fine di un file già esistente, apritelo in accodamento (`"a"`). Per scrivere e leggere in un file, apritelo in aggiornamento in uno dei tre modi disponibili: `"r+"`, `"w+"` o `"a+"`. Il modo `"r+"` apre un file già esistente in lettura e scrittura. Il modo `"w+"` crea un file in lettura e scrittura. Nel caso in cui il file esista già, questo sarà aperto e il suo contenuto corrente sarà eliminato. Il modo `"a+"` apre un file già

Modo	Descrizione
`r`	Apre un file in lettura.
`w`	Crea un file per la scrittura. Nel caso in cui il file esista già, ne eliminerà il contenuto corrente.
`a`	Accoda; apre o crea un file per scrivere alla fine dello stesso.
`r+`	Apre un file in aggiornamento (lettura e scrittura).
`w+`	Crea un file per l'aggiornamento. Nel caso in cui il file esista già, ne eliminerà il contenuto corrente.
`a+`	Accoda; apre o crea un file per l'accodamento; la scrittura sarà eseguita alla fine del file.
`rb`	Apre un file in lettura in modalità binaria.
`wb`	Crea un file per la scrittura in modalità binaria. Nel caso in cui il file esista già, ne eliminerà il contenuto corrente.
`ab`	Accoda; apre o crea un file per scrivere alla fine dello stesso in modalità binaria.
`rb+`	Apre un file in aggiornamento (lettura e scrittura) in modalità binaria.
`wb+`	Crea un file per l'aggiornamento in modalità binaria. Nel caso in cui il file esista già, ne eliminerà il contenuto corrente.
`ab+`	Accoda; apre o crea un file per l'accodamento in modalità binaria; la scrittura sarà eseguita alla fine del file.

Figura 11.6 Modi di apertura dei file

esistente in lettura e in accodamento: tutte le operazioni di scrittura saranno eseguite alla fine del file. Nel caso in cui il file non esista, sarà creato. Si noti che ogni modalità di apertura di un file ha una corrispondente modalità binaria (contenente la lettera `b`) per la manipolazione di file binari. Le modalità binarie verranno utilizzate nelle Sezioni 11.6-11.10 quando introdurremo i file ad accesso casuale.

La funzione `fopen` restituirà un `NULL`, qualora si verifichi un errore durante l'apertura di un file in uno qualsiasi dei modi.

Errore tipico 11.4

Aprire un file inesistente in lettura è un errore.

Errore tipico 11.5

Aprire un file in lettura o in scrittura, senza avere gli appropriati diritti di accesso al file (questo dipende dal sistema operativo) è un errore.

Errore tipico 11.6

Aprire un file in scrittura, quando non c'è più spazio disponibile sul disco è un errore.

Errore tipico 11.7

Aprire un file con il modo di apertura scorretto è un errore logico. Per esempio, aprire un file in scrittura `("w")`*, quando invece dovrebbe essere aperto in aggiornamento* `("r+")`*, provocherà l'eliminazione del contenuto del file.*

Collaudo e messa a punto 11.2

Aprite un file soltanto in lettura (e non in aggiornamento), qualora il suo contenuto non debba essere modificato. Ciò preverrà la modifica non intenzionale dei contenuti del file. Questo è un altro esempio di applicazione del principio del minimo privilegio.

11.5 Leggere i dati da un file ad accesso sequenziale

I dati sono immagazzinati nei file, in modo che possano essere recuperati per l'elaborazione, quando sarà necessario. La sezione precedente ha mostrato come si crea un file ad accesso sequenziale. In questa sezione, discuteremo di come si potranno leggere in modo sequenziale i dati di un file.

Il programma della Figura 11.7 leggerà i record dal file `"clients.dat"` creato dal programma della Figura 11.3 e ne stamperà il contenuto. La riga 11

```
FILE *cfPtr;   /* cfPtr = puntatore al file clients.dat */
```

indica che `cfPtr` sarà un puntatore a `FILE`. La riga 14

```
if ( ( cfPtr = fopen( "clients.dat" ), "r" ) ) == NULL ) {
```

tenterà di aprire in lettura `("r")` il file `"clients.dat"` e determinerà se lo stesso sia stato aperto con successo (ovverosia, controllerà che `fopen` non restituisca un `NULL`). La riga 19

```
fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
```

leggerà un "record" dal file. La funzione `fscanf` è equivalente a `scanf`, eccetto che `fscanf` riceve come argomento un puntatore al file dal quale i dati saranno letti. Una volta che questa istruzione sarà stata eseguita per la prima volta, `account` conterrà il valore `100`, in `name` troveremo `"Jones"` e `balance` varrà `24,98`. Ogni volta che la seconda istruzione `fscanf` (riga 24) sarà stata eseguita, il programma leggerà dal file un altro record e `account`, `name` e `balance` assumeranno dei nuovi valori. Il file sarà chiuso e l'esecuzione del programma sarà terminata (riga 27), una volta che sarà stata raggiunta la fine del file.

```
1  /* Fig. 11.7: fig11_07.c
2     Leggere e stampare un file sequenziale */
3  #include <stdio.h>
4
5  int main()
6  {
7     int account; /* numero del conto */
8     char name[ 30 ]; /* nome dell'intestatario del conto */
9     double balance; /* bilancio del conto */
10
11    FILE *cfPtr;   /* cfPtr = puntatore al file clients.dat */
12
```

Figura 11.7 Leggere e stampare un file sequenziale (continua)

```
13      /* fopen apre il file. Se non riesce a creare il file,
           provoca l'uscita dal programma */
14      if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
15         printf( "File could not be opened\n" );
16      } /* fine del ramo if */
17      else {
18         printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
19         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
20
21         /* finché non si giunge alla fine del file */
22         while ( !feof( cfPtr ) ) {
23            printf( "%-10d%-13s%7.2f\n", account, name, balance );
24            fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
25         } /* fine del comando while */
26
27         fclose(cfPtr); /* fclose chiude il file */
28      } /* fine del ramo else */
29
30      return 0; /* indica che il programma è terminato con successo */
31
32   } /* fine della funzione main */
```

```
Account   Name          Balance
100       Jones           24.98
200       Doe            345.67
300       White            0.00
400       Stone          -42.16
500       Rich           224.62
```

Figura 11.7 Leggere e stampare un file sequenziale

Normalmente, per recuperare in modo sequenziale i dati di un file, un programma incomincerà dall'inizio di quello e leggerà tutti i dati in modo consecutivo finché non siano stati ritrovati quelli desiderati. Durante l'esecuzione di un programma, potrebbe essere utile rielaborare più volte in modo sequenziale i dati di un file, ripartendo dall'inizio dello stesso. Un'istruzione come

```
rewind( cfPtr );
```

fa appunto in modo che il *file position pointer* di un programma, ovverosia il *puntatore di posizione del file* che indica il numero del prossimo byte da leggere o scrivere all'interno di un file, possa essere riposizionato all'inizio (ovverosia al byte 0) del file puntato da `cfPtr`. Il puntatore di posizione del file non è propriamente un puntatore. È piuttosto un valore intero che specifica all'interno del file la posizione del byte in corrispondenza della quale sarà eseguita la prossima lettura o scrittura. A volte è chiamato anche *file offset*. Il puntatore di posizione del file è un membro della struttura `FILE` associata a ogni file.

Il programma della Figura 11.8 consentirà a un credit manager (amministratore del credito) di ottenere gli elenchi dei clienti con un saldo a zero (ovverosia, quelli che non devono alcun denaro), dei clienti con un saldo a credito (vale a dire, quelli ai quali l'azienda deve del

denaro), e dei clienti con un saldo a debito (ovverosia, quelli che devono del denaro all'azienda per beni e servizi ricevuti). Un saldo a credito è un ammontare negativo; mentre quello a debito è un importo positivo.

```
1  /* Fig. 11.8: fig11_08.c
2     Programma per l'interrogazione del credito */
3  #include <stdio.h>
4
5  /* l'esecuzione del programma inizia dalla funzione main */
6  int main()
7  {
8     int request; /* numero della richiesta */
9     int account; /* numero del conto */
10    double balance; /* bilancio del conto */
11    char name[ 30 ]; /* nome dell'intestatario del conto */
12    FILE *cfPtr; /* puntatore al file clients.dat */
13
14    /* fopen apre il file; se non riesce ad aprire il file,
         provoca l'uscita dal programma */
15    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
16       printf( "File could not be opened\n" );
17    } /* fine del ramo if */
18    else {
19
20       /* visualizza le opzioni di richiesta */
21       printf( "Enter request\n"
22               " 1 - List accounts with zero balances\n"
23               " 2 - List accounts with credit balances\n"
24               " 3 - List accounts with debit balances\n"
25               " 4 - End of run\n? " );
26       scanf( "%d", &request );
27
28       /* elabora la richiesta dell'utente */
29       while ( request != 4 ) {
30
31          /* legge il conto, il nome e il bilancio dal file */
32          fscanf(cfPtr, "%d%s%lf", &account, name, &balance);
33
34          switch ( request ) {
35
36             case 1:
37                printf( "\nAccounts with zero balances:\n" );
38
39                /* legge i contenuti del file (finché non
                     viene letto eof) */
40                while ( !feof( cfPtr ) ) {
41
```

Figura 11.8 Programma per l'interrogazione del credito (continua)

```
42                   if ( balance == 0 ) {
43                      printf( "%-10d%-13s%7.2f\n",
44                              account, name, balance );
45                   } /* fine del comando if */
46
47                   /* legge il conto, il nome e il bilancio
                        dal file */
48                   fscanf( cfPtr, "%d%s%lf",
49                           &account, name, &balance );
50                } /* fine del comando while */
51
52                break;
53
54             case 2:
55                printf( "\nAccounts with credit balances:\n" );
56
57                /* legge i contenuti del file (finché non
                     viene letto eof) */
58                while ( !feof( cfPtr ) ) {
59
60                   if ( balance < 0 ) {
61                      printf( "%-10d%-13s%7.2f\n",
62                              account, name, balance );
63                   } /* fine del comando if */
64
65                   /* legge il conto, il nome e il bilancio
                        dal file */
66                   fscanf( cfPtr, "%d%s%lf",
67                           &account, name, &balance );
68                } /* fine del comando while */
69
70                break;
71
72             case 3:
73                printf( "\nAccounts with debit balances:\n" );
74
75                /* legge i contenuti del file (finché non viene
                     letto eof) */
76                while ( !feof( cfPtr ) ) {
77
78                   if ( balance > 0 ) {
79                      printf( "%-10d%-13s%7.2f\n",
80                              account, name, balance );
81                   } /* fine del comando if */
82
83                   /* legge il conto, il nome e il bilancio
                        dal file */
84                   fscanf( cfPtr, "%d%s%lf",
85                           &account, name, &balance );
```

Figura 11.8 Programma per l'interrogazione del credito (continua)

```
86                  } /* fine del comando while */
87
88                  break;
89
90              } /* fine del comando switch */
91
92              rewind( cfPtr ); /* riporta cfPtr all'inizio del file */
93
94              printf( "\n? " );
95              scanf( "%d", &request );
96           } /* fine del comando while */
97
98           printf( "End of run.\n" );
99           fclose(cfPtr); /* fclose chiude il file */
100       } /* fine del ramo else */
101
102       return 0; /* indica che il programma è terminato
                       con successo */
103
104    } /* fine della funzione main */
```

Figura 11.8 Programma per l'interrogazione del credito

Il programma visualizzerà un menu e consentirà al credit manager di immettere una delle tre scelte disponibili per ottenere le informazioni sul credito. La scelta 1 produrrà l'elenco dei conti con un saldo a zero. La scelta 2 produrrà l'elenco dei conti con un saldo a credito. La scelta 3 produrrà l'elenco dei conti con un saldo a debito. La scelta 4 terminerà l'esecuzione del programma. Un output di esempio è mostrato nella Figura 11.9.

Osservate che in questo tipo di file sequenziale i dati non potranno essere modificati, senza correre il rischio di distruggere gli altri dati del file. Per esempio, se fosse necessario cambiare `"White"` in `"Worthington"`, il vecchio nome non potrà essere semplicemente sostituito. Il record per `White` è stato scritto nel file come

```
300 White 0.00
```

Usando il nuovo nome e partendo dalla stessa posizione all'interno del file, se il record fosse stato sostituito sarebbe diventato

```
300 Worthington 0.00
```

Il nuovo record è più lungo di quello originale e, di conseguenza, i caratteri successivi alla seconda `"o"` di `"Worthington"` avrebbero cancellato la parte iniziale del record sequenziale successivo. Il problema, in questo caso, è che nel *modello formattato di input/output* usato con `fprintf` e `fscanf`, i campi, e quindi i record, possono cambiare la loro dimensione. Per esempio, 7, 14, –117, 2074 e 27383 sono tutti `int` immagazzinati internamente nello stesso numero di byte, ma sono campi di dimensioni differenti nel momento in cui vengono visualizzati sullo schermo o vengono scritti in un file sotto forma testuale.

Ne consegue che, di norma, per aggiornare in modo appropriato i record non si userà un accesso sequenziale con `fprintf` e `fscanf`, ma si preferirà riscrivere l'intero file. In un file ad accesso sequenziale come quello del nostro esempio, per cambiare il suddetto nome i record che precedono `300 White 0.00` dovranno essere ricopiati in un nuovo file al quale, a questo

punto, si dovrà aggiungere il record modificato e infine quelli successivi a `300 White 0.00`. Tutto ciò significa che, per aggiornare un solo record dovranno essere elaborati anche tutti gli altri.

```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300        White            0.00

? 2

Accounts with credit balances:
400        Stone          -42.16

? 3

Accounts with debit balances:
100        Jones           24.98
200        Doe            345.67
500        Rich           224.62

? 4
End of run.
```

Figura 11.9 L'output di esempio del programma per l'interrogazione del credito mostrato nella Figura 11.8

11.6 I file ad accesso casuale

Come abbiamo affermato in precedenza, i record di un file creati con l'output formattato della funzione `fprintf` non avranno necessariamente la stessa lunghezza. Ogni record di un *file ad accesso casuale*, invece, avrà normalmente una lunghezza fissa e vi si potrà accedere in modo diretto (e quindi velocemente), senza passare attraverso gli altri record. Ciò fa sì che i file ad accesso casuale siano appropriati per i sistemi di prenotazione delle linee aeree, per quelli bancari, per quelli destinati alla gestione di un punto vendita, e per altri tipi di *sistemi per l'elaborazione di transazioni* che richiedano un accesso rapido a dati specifici. Esistono altri modi di implementare i file ad accesso casuale, ma noi limiteremo la nostra discussione a un approccio lineare, usando i record a lunghezza fissa.

Dato che in un file ad accesso casuale ogni record ha normalmente la stessa lunghezza, la sua esatta posizione, relativamente all'inizio del file, potrà essere calcolata con una funzione della chiave del record. Vedremo presto come questo faciliti l'accesso immediato a record specifici, anche in file di grandi dimensioni.

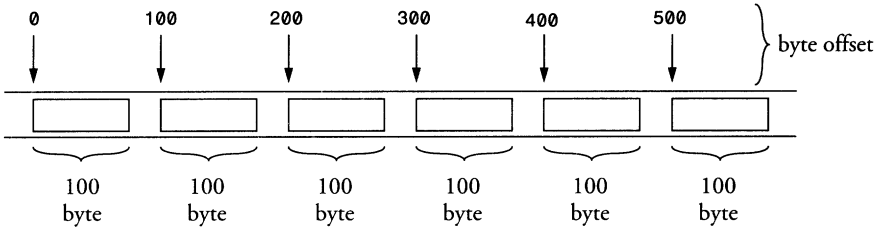


Figura 11.10 Rappresentazione di un file ad accesso casuale in C

La Figura 11.10 illustra un modo di implementare un file ad accesso casuale. Un file di questo genere è come un treno merci con molti vagoni, alcuni vuoti e altri con un carico. Ogni vagone del treno ha la stessa lunghezza.

In un file ad accesso casuale, i nuovi dati potranno essere inseriti senza distruggere quelli già immagazzinati. I dati immagazzinati in precedenza potranno anche essere aggiornati o eliminati, senza che si debba riscrivere l'intero file. Nelle prossime sezioni, spiegheremo come creare un file ad accesso casuale, immettervi dei dati, leggerli in modo sequenziale o casuale, aggiornarli ed eliminarli quando non saranno più necessari.

11.7 Creare un file ad accesso casuale

La funzione `fwrite` trasferisce in un file un numero specificato di byte, partendo da una data posizione di memoria. I dati sono scritti nel file, partendo dal byte indicato dal puntatore di posizione del file. La funzione `fread` trasferisce un numero specificato di byte, dalla posizione indicata dal file offset a un'area di memoria che incomincia a un indirizzo specificato. A questo punto, per scrivere un intero, invece di usare

```
fprintf( fPtr, "%d", number );
```

che potrebbe stampare da un minimo di una a un massimo di 11 cifre (10 numeri più il segno, ognuno dei quali richiede un byte di memoria), per un intero di quattro byte, potremmo usare

```
fwrite( &number, sizeof( int ), 1, fPtr );
```

che scriverebbe in ogni caso quattro byte (o due, su un sistema con interi di due byte) dalla variabile `number` al file rappresentato da `fPtr` (l'argomento `1` sarà spiegato tra breve). In seguito, `fread` potrà essere usata per leggere quattro di quei byte nella variabile intera `number`. Se da un lato `fread` e `fwrite` leggono e scrivono dei dati come gli interi in un formato con dimensione fissa, invece che variabile, dall'altro i dati che esse gestiscono sono elaborati nel formato "raw data" (dato grezzo) del computer (ovverosia, in byte di dati), piuttosto che nel formato umanamente intelligibile di `printf` e `scanf`.

Le funzioni `fwrite` e `fread` sono in grado di leggere e scrivere su disco interi vettori di dati. Il terzo argomento di `fwrite` e `fread` è appunto il numero degli elementi del vettore che dovranno essere letti dal disco, o che vi dovranno essere scritti. La precedente chiamata della funzione `fwrite` scriverà un singolo intero su disco, perciò il terzo argomento è `1` (come se stessimo scrivendo un solo elemento di un vettore).

È raro che i programmi per l'elaborazione dei file scrivano un solo campo all'interno di un file. Normalmente, vi scriveranno una `struct` per volta, come mostrato negli esempi seguenti.

Considerate il seguente enunciato di problema:

> *Create un sistema per l'elaborazione del credito che sia in grado di memorizzare fino a 100 record di lunghezza fissa. Ogni record dovrà consistere di un numero di conto, che sarà usato come chiave del record, un cognome, un nome e un saldo. Il programma risultante dovrà essere in grado di aggiornare un conto, inserirne uno nuovo, eliminarne uno già esistente ed elencare tutti i record di conto in un file di testo formattato per la visualizzazione. Usate un file ad accesso casuale.*

Le prossime sezioni presenteranno le tecniche necessarie per creare il suddetto programma per l'elaborazione del credito. Il programma della Figura 11.11 mostra come aprire un file ad accesso casuale, definire un formato di record usando una `struct`, scrivere i dati sul disco e chiudere il file. Questo programma inizializzerà tutti i 100 record del file `"credit.dat"` con la `struct` vuota, usando la funzione `fwrite`. Ogni `struct` vuota contiene `0` per il numero di conto, `""` (la stringa vuota) per il cognome, `""` per il nome, e `0,0` per il saldo. Il file sarà inizializzato in questo modo per creare sul disco lo spazio nel quale sarà immagazzinato, e per far sì che si possa determinare se un record contenga dei dati.

```
1  /* Fig. 11.11: fig11_11.c
2     Creare in modo sequenziale un file ad accesso casuale */
3  #include <stdio.h>
4
5  /* dichiarazione della struttura clientData */
6  struct clientData {
7     int acctNum; /* numero del conto */
8     char lastName[ 15 ]; /* cognome dell'intestatario del conto */
9     char firstName[ 10 ]; /* nome dell'intestatario del conto */
10    double balance; /* bilancio del conto */
11 }; /* fine della struttura clientData */
12
13 int main()
14 {
15    int i; /* contatore usato per contare da 1 a 100 */
16
17    /* crea clientData con informazioni di default */
18    struct clientData blankClient = { 0, "", "", 0.0 };
19
20    FILE *cfPtr; /* puntatore al file credit.dat */
21
22    /* fopen apre il file; se non riesce ad aprire il file,
         provoca l'uscita dal programma */
23    if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24       printf( "File could not be opened.\n" );
25    } /* fine del ramo if */
26    else {
```

Figura 11.11 Creare in modo sequenziale un file ad accesso casuale (continua)

```
27
28          /* manda in output sul file 100 record vuoti */
29          for ( i = 1; i <= 100; i++ ) {
30             fwrite( &blankClient, sizeof(struct clientData), 1,
                    cfPtr );
31          } /* fine del comando for */
32
33          fclose ( cfPtr ); /* fclose chiude il file */
34       } /* fine del ramo else */
35
36       return 0; /* indica che il programma è terminato con successo */
37
38    } /* fine della funzione main */
```

Figura 11.11 Creare in modo sequenziale un file ad accesso casuale

La funzione `fwrite` scrive un blocco (un numero specificato di byte) di dati in un file. In questo programma, la riga 30

```
fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
```

scriverà sul file puntato da `cfPtr` la struttura `blankClient` di lunghezza `sizeof( struct clientData )`. L'operatore `sizeof` restituisce la dimensione in byte dell'operando contenuto nelle parentesi (in questo caso `struct clientData`). L'operatore `sizeof` restituisce un intero senza segno e può essere usato per determinare le dimensioni in byte di un qualsiasi tipo di dato o di espressione. Per esempio, `sizeof( int )` è usato per determinare se un intero è immagazzinato in due o quattro byte, su un particolare computer.

La funzione `fwrite` può davvero essere usata per scrivere vari elementi di un vettore di oggetti. Per farlo, il programmatore dovrà fornire come primo argomento della chiamata a `fwrite` un puntatore a un vettore e, come terzo argomento della stessa chiamata, dovrà specificare il numero di elementi che saranno scritti. Nell'istruzione precedente, `fwrite` è stata usata per scrivere un oggetto singolo che non era un membro di un vettore. Scrivere un oggetto singolo equivale alla scrittura di un solo elemento di un vettore, da cui il numero `1` nella chiamata a `fwrite`.

11.8 Scrivere i dati in modo casuale in un file ad accesso casuale

Il programma della Figura 11.12 scriverà i propri dati nel file `"credit.dat"`. Esso combinerà l'utilizzo di `fseek` e `fwrite`, per immagazzinare i dati in posizioni specifiche del file. La funzione `fseek` imposterà il file offset in una posizione specifica e, in seguito, `fwrite` vi scriverà i dati. Nella Figura 11.13, è illustrato un esempio di esecuzione.

```
1   /* Fig. 11.12: fig11_12.c
2      Scrivere in un file ad accesso casuale */
3   #include <stdio.h>
4
5   /* dichiarazione della struttura clientData */
```

Figura 11.12 Scrivere i dati in modo casuale in un file ad accesso casuale (continua)

```
 6  struct clientData {
 7     int acctNum; /* numero del conto */
 8     char lastName[ 15 ]; /* cognome dell'intestatario del conto */
 9     char firstName[ 10 ]; /* nome dell'intestatario del conto */
10     double balance; /* bilancio del conto */
11  }; /* fine della struttura clientData */
12
13  int main()
14  {
15     FILE *cfPtr; /* puntatore al file credit.dat */
16
17     /* crea clientData con informazioni di default */
18     struct clientData client = { 0, "", "", 0.0 };
19
20     /* fopen apre il file; se non riesce ad aprire il file,
          provoca l'uscita dal programma */
21     if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
22        printf( "File could not be opened.\n" );
23     } /* fine del ramo if */
24     else {
25
26        /* chiede all'utente di specificare il numero del conto */
27        printf( "Enter account number"
28                " ( 1 to 100, 0 to end input )\n? " );
29        scanf( "%d", &client.acctNum );
30
31        /* l'utente inserisce l'informazione che viene copiata
             nel file */
32        while ( client.acctNum != 0 ) {
33
34           /* l'utente inserisce il cognome, il nome e il bilancio */
35           printf( "Enter lastname, firstname, balance\n? " );
36
37           /* imposta il cognome, il nome e il valore del bilancio
              del record */
38           fscanf( stdin, "%s%s%lf", client.lastName,
39                 client.firstName, &client.balance );
40
41           /* trova la posizione nel file del record specificato
                dall'utente */
42           fseek( cfPtr, ( client.acctNum - 1 ) *
43                 sizeof( struct clientData), SEEK_SET );
44
45           /* scrive l'informazione specificata dall'utente
                nel file */
46           fwrite( &client, sizeof( struct clientData ), 1, cfPtr );
47
48           /* consente all'utente di specificare un altro
                numero di conto */
```

Figura 11.12 Scrivere i dati in modo casuale in un file ad accesso casuale (continua)

```
49            printf( "Enter account number\n? " );
50            scanf( "%d", &client.acctNum );
51         } /* fine del comando while */
52
53         fclose(cfPtr); /* fclose chiude il file */
54      } /* fine del ramo else */
55
56      return 0; /* indica che il programma è terminato con successo */
57
58   } /* fine della funzione main */
```

Figura 11.12 Scrivere i dati in modo casuale in un file ad accesso casuale

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Figura 11.13 Esempio di esecuzione del programma in Figura 11.12

Le righe 42-43

```
fseek( cfPtr, ( client.acctNum -1 ) *
     sizeof( struct clientData ), SEEK_SET );
```

sistemeranno il puntatore di posizione del file puntato da `cfPtr` sul byte calcolato da `( client.acctNum - 1 ) * sizeof( struct clientData )`. Il valore di questa espressione è detto *offset* o *scostamento*. Dato che i numeri di conto sono compresi tra uno e 100, mentre le posizioni in byte all'interno del file incominciano da `0`, si dovrà sottrarre 1 dal numero di conto quando si dovrà calcolare la posizione in byte del record. Per il primo record, quindi, il puntatore di posizione del file sarà impostato sul byte zero. La costante simbolica `SEEK_SET` indica che il puntatore di posizione del file dovrà essere sistemato, relativamente al suo inizio, in base al

valore dell'offset. Come indicato dalla precedente istruzione, la ricerca del conto numero uno imposterà all'inizio del file il relativo puntatore di posizione, perché il suo valore calcolato in byte sarà zero. La Figura 11.14 mostra il puntatore del file mentre fa riferimento alla struttura `FILE` in memoria. Il puntatore di posizione del file indica che il prossimo byte da leggere o scrivere si trova a cinque byte dall'inizio del file.

Il prototipo di funzione per la `fseek` è

```
int fseek( FILE *stream, long int offset, int whence );
```

dove `offset` è il numero di byte da ricercare dalla posizione `whence` nel file puntato da `stream`. L'argomento `whence` potrà contenere uno dei tre valori `SEEK_SET`, `SEEK_CUR` o `SEEK_END` (tutti definiti in `<stdio.h>`), che indicano la posizione del file da cui inizia la ricerca: `SEEK_SET` indica che questa incomincerà dall'inizio del file; `SEEK_CUR` indica che la ricerca incomincerà dalla posizione corrente del file; mentre `SEEK_END` indica che la ricerca inizierà dalla fine del file.

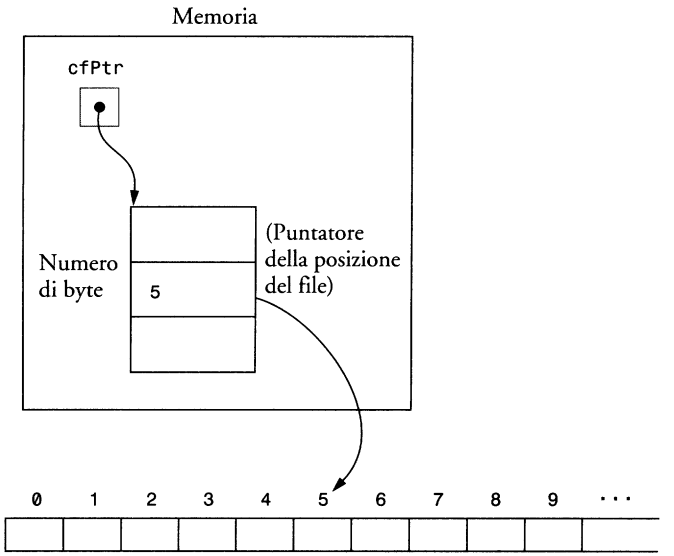


Figura 11.14 Il puntatore di posizione del file mentre indica un offset di cinque byte dall'inizio del file

11.9 Leggere i dati in modo casuale da un file ad accesso casuale

La funzione `fread` legge uno specificato numero di byte da un file e li immagazzina nella memoria. Per esempio, l'istruzione

```
fread( &client, sizeof( struct clientData ), 1, cfPtr );
```

leggerà il numero di byte determinato da `sizeof( struct clientData )` dal file puntato da `cfPtr` e immagazzinerà i dati nella struttura `client`. I byte saranno letti dalla posizione specificata dal puntatore di posizione del file. La funzione `fread` potrà essere usata per leggere degli elementi a lunghezza fissa in un vettore, fornendo un puntatore a quest'ultimo e indicando quanti ne dovranno essere letti. L'istruzione precedente specifica che dovrà essere

letto un solo elemento. Per leggere più di un elemento, dovrete indicarne il numero nel terzo argomento dell'istruzione `fread`.

Il programma della Figura 11.15 leggerà in modo sequenziale tutti i record del file `"credit.dat"`, determinerà se contengano dei dati e, in tal caso, li visualizzerà in modo formattato. La funzione `feof` determinerà se sia stata raggiunta la fine del file, mentre `fread` trasferirà i dati dal disco alla struttura `client` di tipo `clientData`.

```
1  /* Fig. 11.15: fig11_15.c
2     Leggere in modo sequenziale un file ad accesso casuale */
3  #include <stdio.h>
4
5  /* dichiarazione della struttura clientData */
6  struct clientData {
7     int acctNum; /* numero del conto */
8     char lastName[ 15 ]; /* cognome dell'intestatario del conto */
9     char firstName[ 10 ]; /* nome dell'intestatario del conto */
10    double balance; /* bilancio del conto */
11 }; /* fine della struttura clientData */
12
13 int main()
14 {
15    FILE *cfPtr; /* puntatore al file credit.dat */
16
17    /* crea clientData con informazioni di default */
18    struct clientData client = { 0, "", "", 0.0 };
19
20    /* fopen apre il file; se non riesce ad aprire il file,
         provoca l'uscita dal programma */
21    if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
22       printf( "File could not be opened.\n" );
23    } /* fine del ramo if */
24    else {
25       printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
26             "First Name", "Balance" );
27
28       /* legge tutti i record dal file (finché non viene
            letto eof) */
29       while ( !feof( cfPtr ) ) {
30          fread( &client, sizeof( struct clientData ), 1, cfPtr );
31
32          /* visualizza il record */
33          if ( client.acctNum != 0 ) {
34             printf( "%-6d%-16s%-11s%10.2f\n",
35                   client.acctNum, client.lastName,
36                   client.firstName, client.balance );
37          } /* fine del comando if */
38
39       } /* fine del comando while */
```

Figura 11.15 Leggere in modo sequenziale un file ad accesso casuale (continua)

```
40
41          fclose(cfPtr); /* fclose chiude il file */
42       } /* fine del ramo else */
43
44       return 0; /* indica che il programma è terminato con successo */
45
46    } /* fine della funzione main */
```

```
Acct  Last Name      First Name    Balance
29    Brown          Nancy          -24.54
33    Dunn           Stacey         314.33
37    Barker         Doug             0.00
88    Smith          Dave           258.34
96    Stone          Sam             34.98
```

Figura 11.15 Leggere in modo sequenziale un file ad accesso casuale

11.10 Studio di un caso: un programma per l'elaborazione delle transazioni

Presentiamo ora un corposo programma per l'elaborazione delle transazioni, che utilizzerà i file ad accesso casuale. Il programma gestirà le informazioni relative ai conti di una banca. Esso aggiornerà i conti esistenti, aggiungerà quelli nuovi, li eliminerà e archivierà un elenco di tutti i conti correnti in un file di testo idoneo alla visualizzazione. Presumeremo che il programma della Figura 11.11 sia già stato eseguito e che abbia già creato il file `credit.dat`.

Il programma ha cinque scelte. La scelta uno richiamerà la funzione `textFile` per immagazzinare un elenco formattato di tutti i conti in un file di testo, chiamato `accounts.txt`, che potrà essere visualizzato in un secondo momento. La funzione userà `fread` e le tecniche di accesso sequenziale ai file, utilizzate nel programma della Figura 11.15. Dopo aver selezionato la prima scelta del menu, il file `accounts.txt` conterrà:

```
Acct  Last Name      First Name    Balance
29    Brown          Nancy          -24.54
33    Dunn           Stacey         314.33
37    Barker         Doug             0.00
88    Smith          Dave           258.34
96    Stone          Sam             34.98
```

La scelta due richiamerà la funzione `updateRecord` per aggiornare un conto. Questa aggiornerà soltanto un conto già esistente e perciò dovrà controllare, in primo luogo, che il record specificato dall'utente non sia vuoto. Il record sarà letto con `fread`, immagazzinato nella struttura `client` e, successivamente, il suo membro `acctNum` sarà confrontato con uno zero. Qualora in quel membro fosse ritrovato uno zero, sapremo che il record non contiene informazioni e, di conseguenza, visualizzeremo un messaggio per indicare che il record è vuoto. In seguito, saranno visualizzate nuovamente le scelte del menu. Invece, qualora il record contenesse delle informazioni, la funzione `updateRecord` accetterà in input l'ammontare della transazione, calcolerà il nuovo saldo e riscriverà il record nel file. Un output tipico della seconda scelta di menu sarà:

```
Enter account to update (1 - 100): 37
37    Barker          Doug            0.00

Enter charge (+) or payment (-): +87.99
37    Barker          Doug            87.99
```

La scelta tre richiamerà la funzione `newRecord` per aggiungere un nuovo conto al file. Qualora l'utente abbia immesso un numero di conto corrispondente a un record già esistente, `newRecord` visualizzerà un messaggio di errore, indicando che il record contiene già delle informazioni, e le scelte del menu saranno visualizzate nuovamente. Per aggiungere un nuovo record, questa funzione utilizzerà lo stesso processo usato dal programma della Figura 11.12. Un tipico output della terza scelta di menu sarà:

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

La scelta quattro richiamerà la funzione `deleteRecord` per eliminare un record dal file. L'eliminazione sarà eseguita chiedendo all'utente il numero di conto e inizializzando nuovamente il record corrispondente. Qualora il conto indicato non contenesse delle informazioni, `deleteRecord` visualizzerà un messaggio di errore indicando che il conto non esiste. La quinta scelta di menu terminerà l'esecuzione del programma. Nella Figura 11.16 è mostrato il programma sinora descritto. Osservate che il file `"credit.dat"` sarà aperto in aggiornamento (lettura e scrittura), usando il modo `"rb+"`.

```
1   /* Fig. 11.16: fig11_16.c
2      aggiorna i dati già scritti nel file, crea nuovi dati da inserire
3      nel file, ed elimina i dati già presenti nel file. */
4
5   #include <stdio.h>
6
7   /* dichiarazione della struttura clientData */
8   struct clientData {
9      int acctNum; /* numero del conto */
10     char lastName[ 15 ]; /* cognome dell'intestatario del conto */
11     char firstName[ 10 ]; /* nome dell'intestatario del conto */
12     double balance; /* bilancio del conto */
13  }; /* fine della struttura clientData */
14
15  /* prototipi */
16  int enterChoice( void );
17  void textFile( FILE *readPtr );
18  void updateRecord( FILE *fPtr );
19  void newRecord( FILE *fPtr );
20  void deleteRecord(FILE *fPtr );
21
22  int main()
23  {
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
24       FILE *cfPtr; /* puntatore al file credit.dat */
25       int choice; /* scelta dell'utente */
26
27       /* fopen apre il file; se non riesce ad aprire il file,
            provoca l'uscita dal programma */
28       if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
29          printf( "File could not be opened.\n" );
30       } /* fine del ramo if */
31       else {
32
33          /* permette all'utente di specificare un'azione */
34          while ( ( choice = enterChoice() ) != 5 ) {
35
36             switch ( choice ) {
37
38                /* crea il file di testo a partire dal file
                     dei record */
39                case 1:
40                   textFile( cfPtr );
41                   break;
42
43                /* aggiorna il record */
44                case 2:
45                   updateRecord( cfPtr );
46                   break;
47
48                /* crea il record */
49                case 3:
50                   newRecord( cfPtr );
51                   break;
52
53                /* cancella un record esistente */
54                case 4:
55                   deleteRecord( cfPtr );
56                   break;
57
58                /* visualizza un messaggio se l'utente non specifica
                     una scelta valida */
59                default:
60                   printf( "Incorrect choice\n" );
61                   break;
62
63             } /* fine del comando switch */
64
65          } /* fine del comando while */
66
67          fclose( cfPtr ); /* fclose chiude il file */
68       } /* fine del ramo else */
69
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
70       return 0; /* indica che il programma è terminato con successo */
71
72    } /* fine della funzione main */
73
74    /* crea un file di testo formattato per la stampa */
75    void textFile( FILE *readPtr )
76    {
77       FILE *writePtr; /* puntatore al file accounts.txt */
78
79       /* crea clientData con informazioni di default */
80       struct clientData client = { 0, "", "", 0.0 };
81
82       /* fopen apre il file; se non riesce ad aprire il file,
            provoca l'uscita dalla funzione */
83       if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL ) {
84          printf( "File could not be opened.\n" );
85       } /* fine del ramo if */
86       else {
87          rewind( readPtr ); /* imposta il puntatore all'inizio
                                del file */
88          fprintf( writePtr, "%-6s%-16s%-11s%10s\n",
89                   "Acct", "Last Name", "First Name","Balance" );
90
91          /* copia tutti i record dal file ad accesso casuale
               nel file di testo */
92          while ( !feof( readPtr ) ) {
93             fread( &client, sizeof( struct clientData ), 1, readPtr );
94
95             /* scrive un singolo record nel file di testo */
96             if ( client.acctNum != 0 ) {
97                fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
98                         client.acctNum, client.lastName,
99                         client.firstName, client.balance );
100            } /* fine del comando if */
101
102         } /* fine del comando while */
103
104         fclose( writePtr ); /* fclose chiude il file */
105      } /* fine del ramo else */
106
107   } /* fine della funzione textFile */
108
109   /* aggiorna il bilancio nel record */
110   void updateRecord( FILE *fPtr )
111   {
112      int account; /* numero del conto */
113      double transaction; /* ammontare della transazione */
114
115      /* crea clientData con informazioni di default */
116      struct clientData client = { 0, "", "", 0.0 };
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
117
118      /* ottiene il numero del conto da aggiornare */
119      printf( "Enter account to update ( 1 - 100 ): " );
120      scanf( "%d", &account );
121
122      /* sposta il puntatore del file nel punto del record
            giusto nel file */
123      fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
124            SEEK_SET );
125
126      /* legge il record dal file */
127      fread( &client, sizeof( struct clientData ), 1, fPtr );
128
129      /* visualizza un messaggio d'errore se il conto non esiste */
130      if ( client.acctNum == 0 ) {
131         printf( "Account #%d has no information.\n", account );
132      } /* fine del ramo if */
133      else { /* aggiorna il record */
134         printf( "%-6d%-16s%-11s%10.2f\n\n",
135                client.acctNum, client.lastName,
136                client.firstName, client.balance );
137
138         /* richiede all'utente l'ammontare della transazione */
139         printf( "Enter charge ( + ) or payment ( - ): " );
140         scanf("%f", &transaction);
141         client.balance += transaction; /* aggiorna il bilancio
                                              del record */
142
143         printf("%-6d%-16s%-11s%10.2f\n",
144                client.acctNum, client.lastName,
145                client.firstName, client.balance);
146
147         /* sposta il puntatore del file per correggere
               il record nel file */
148         fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
149               SEEK_SET );
150
151         /* sovrascrive il vecchio record con il record
               aggiornato nel file */
152         fwrite( &client, sizeof( struct clientData ), 1, fPtr );
153      } /* fine del ramo else */
154
155   } /* fine della funzione updateRecord */
156
157   /* cancella un record esistente */
158   void deleteRecord( FILE *fPtr )
159   {
160
161      struct clientData client; /* memorizza il record letto
                                     dal file */
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
162      struct clientData blankClient = {0, "", "", 0}; /* cliente
                                                            vuoto */
163
164      int accountNum; /* numero del conto */
165
166      /* ottiene il numero del conto da cancellare */
167      printf( "Enter account number to delete ( 1 - 100 ): " );
168      scanf( "%d", &accountNum );
169
170      /* sposta il puntatore del file nel punto del record
            giusto nel file */
171      fseek( fPtr, ( accountNum - 1 ) * sizeof( struct clientData ),
172           SEEK_SET );
173
174      /* legge il record dal file */
175      fread( &client, sizeof( struct clientData ), 1, fPtr );
176
177      /* visualizza un messaggio d'errore se il conto non esiste */
178      if (client.acctNum == 0)  {
179         printf( "Account %d does not exist.\n", accountNum );
180      } /* fine del ramo if */
181      else { /* cancella il record */
182
183         /* sposta il puntatore del file nel punto del record
               giusto nel file */
184         fseek( fPtr, ( accountNum - 1 ) * sizeof( struct
                                          clientData ),
185              SEEK_SET );
186
187         /* sostituisce il record esistente con un record vuoto */
188         fwrite( &blankClient,
189              sizeof( struct clientData ), 1, fPtr );
190      } /* fine del ramo else */
191
192   } /* fine della funzione deleteRecord */
193
194   /* crea e inserisce un record */
195   void newRecord( FILE *fPtr )
196   {
197      /* crea clientData con informazioni di default */
198      struct clientData client = { 0, "", "", 0.0 };
199
200      int accountNum; /* numero del conto */
201
202      /* ottiene il numero del conto da creare */
203      printf( "Enter new account number ( 1 - 100 ): " );
204      scanf( "%d", &accountNum );
205
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
206      /* sposta il puntatore del file nel punto del record
            giusto nel file */
207      fseek( fPtr, ( accountNum - 1 ) * sizeof( struct
                                          clientData ),
208           SEEK_SET );
209
210      /* legge il record dal file */
211      fread( &client, sizeof( struct clientData ), 1, fPtr );
212
213      /* visualizza un messaggio d'errore se il conto esiste già */
214      if ( client.acctNum != 0 ) {
215         printf( "Account #%d already contains information.\n",
216                client.acctNum );
217      } /* fine del ramo if */
218      else { /* crea il record */
219
220         /* l'utente inserisce il cognome, il nome e il bilancio */
221         printf( "Enter lastname, firstname, balance\n? " );
222         scanf( "%s%s%lf", &client.lastName, &client.firstName,
223               &client.balance );
224
225         client.acctNum = accountNum;
226
227         /* sposta il puntatore del file nel punto del record
               giusto nel file */
228         fseek( fPtr, ( client.acctNum - 1 ) *
229               sizeof( struct clientData ), SEEK_SET );
230
231         /* inserisce il record nel file */
232         fwrite( &client,
233              sizeof( struct clientData ), 1, fPtr );
234      } /* fine del ramo else */
235
236   } /* fine della funzione newRecord */
237
238   /* permette all'utente di specificare una scelta del menu */
239   int enterChoice( void )
240   {
241      int menuChoice; /* variabile che memorizza la scelta
                          dell'utente */
242
243      /* visualizza le opzioni disponibili */
244      printf( "\nEnter your choice\n"
245            "1 - store a formatted text file of accounts called\n"
246            "    \"accounts.txt\" for printing\n"
247            "2 - update an account\n"
248            "3 - add a new account\n"
249            "4 - delete an account\n"
250            "5 - end program\n? " );
```

Figura 11.16 Il programma per la gestione del conto bancario (continua)

```
251
252       scanf( "%d", &menuChoice ); /* riceve la scelta da parte
                                         dell'utente */
253
254       return menuChoice;
255
256    } /* fine della funzione enterChoice */
```

Figura 11.16 Il programma per la gestione del conto bancario

Esercizi di autovalutazione

11.1 Riempite gli spazi in ognuna delle seguenti righe:

a) In ultima analisi, tutte le unità di informazione elaborate da un computer sono ridotte a combinazioni di _____ e di _____.
b) La più piccola unità di informazione che un computer può elaborare è il _____.
c) Un _____ è un gruppo di record correlati.
d) Numeri, lettere e simboli speciali sono complessivamente chiamati _____.
e) Un gruppo di file correlati è un _____.
f) La funzione _____ chiude un file.
g) L'istruzione _____ legge i dati da un file, in modo simile a quello in cui `scanf` legge dallo `stdin`.
h) La funzione _____ legge un carattere da un file specificato.
i) La funzione _____ legge una riga da un file specificato.
j) La funzione _____ apre un file.
k) La funzione _____ è normalmente usata, nelle applicazioni ad accesso casuale, per leggere i dati da un file.
l) La funzione _____ riposiziona il file offset su una posizione specifica.

11.2 Determinate quali delle seguenti affermazioni siano *vere* e quali *false*. Per quelle che ritenete false, spiegatene il motivo.

a) La funzione `fscanf` non può essere usata per leggere i dati dallo standard input.
b) Il programmatore deve usare esplicitamente la `fopen`, per aprire gli stream dello standard input, standard output e standard error.
c) Per chiudere un file, un programma deve richiamare esplicitamente la funzione `fclose`.
d) Qualora il puntatore di posizione di un file sequenziale faccia riferimento a un byte diverso dall'inizio del file, questo dovrà essere chiuso e riaperto per leggere nuovamente da quel punto.
e) La funzione `fprintf` può scrivere sullo standard output.
f) Si potrà sempre aggiornare un dato in un file ad accesso sequenziale, senza rischiare di cancellare gli altri dati.
g) Per trovare un record specifico in un file ad accesso casuale, non è necessario scorrere tutti gli altri record.
h) I record dei file ad accesso casuale non hanno una lunghezza uniforme.
i) La funzione `fseek` può ricercare soltanto dall'inizio di un file.

11.3 Scrivete una singola istruzione, per svolgere ognuno dei seguenti compiti. Supponete che ognuna di queste istruzioni debba essere implementata nello stesso programma:

a) Scrivete un'istruzione che apra in lettura il file `"oldmast.dat"` e assegni a `ofPtr` il puntatore ottenuto.
b) Scrivete un'istruzione che apra in lettura il file `"trans.dat"` e assegni a `tfPtr` il puntatore ottenuto.
c) Scrivete un'istruzione che apra in scrittura (e creazione) il file `"newmast.dat"` e assegni a `nfPtr` il puntatore ottenuto.

d) Scrivete un'istruzione che legga un record dal file `"oldmast.dat"`. Il record sarà formato dall'intero `accountNum`, dalla stringa `name` e dal valore in virgola mobile `currentBalance`.
e) Scrivete un'istruzione che legga un record dal file `"trans.dat"`. Il record sarà formato dall'intero `accountNum` e dal valore in virgola mobile `dollarAmount`.
f) Scrivete un'istruzione che registri un record nel file `"newmast.dat"`. Il record sarà formato dall'intero `accountNum`, dalla stringa `name` e dal valore in virgola mobile `currentBalance`.

11.4 Trovate l'errore in ognuno dei seguenti frammenti di programma. Spiegate come possa essere corretto:

a) Il file puntato da `fPtr` (`"payables.dat"`) non è stato aperto.

```
fprintf( fPtr, "%d%s%d\n", account, company, amount );
```

b) `open( "receive.dat","r+" );`
c) L'istruzione seguente dovrebbe leggere un record da `"payables.dat"`. Il puntatore `payPtr` fa riferimento a questo file, mentre il puntatore `recPtr` fa riferimento a `"receive.dat"`.

```
fscanf( recPtr, "%d%s%d\n", &account, company, &amount );
```

d) Il file `"tools.dat"` dovrebbe essere aperto per aggiungervi dei dati, senza eliminare quelli correnti.

```
if ( ( tfPtr = fopen( "tools.dat", "w" ) ) != NULL )
```

e) Il file `"courses.dat"` dovrebbe essere aperto in accodamento, senza modificare il contenuto corrente del file.

```
if ( ( cfPtr = fopen( "courses.dat", "w+" ) ) != NULL )
```

Risposte agli esercizi di autovalutazione

11.1 a) 1, 0; b) Bit; c) File; d) Caratteri; e) Database; f) `fclose`; g) `fscanf`; h) `fgetc`; i) `fgets`; j) `fopen`; k) `fread`; l) `fseek`.

11.2 a) Falso. La funzione `fscanf` può essere usata per leggere dallo standard input includendo il puntatore allo stream corrispondente, `stdin`, nella chiamata a `fscanf`.
b) Falso. Questi tre stream saranno aperti automaticamente dal C, quando comincerà l'esecuzione del programma.
c) Falso. I file saranno chiusi quando sarà terminata l'esecuzione del programma, tuttavia, tutti i file dovrebbero essere chiusi esplicitamente con `fclose`.
d) Falso. Può essere usata la funzione `rewind` per riposizionare all'inizio del file il relativo puntatore di posizione.
e) Vero.
f) Falso. Nella maggior parte dei casi i record dei file sequenziali non hanno una lunghezza uniforme. Di conseguenza, è probabile che l'aggiornamento di un record possa causare la cancellazione di altri dati.
g) Vero.
h) Falso. I record dei file ad accesso casuale hanno normalmente una lunghezza uniforme.
i) Falso. È possibile cercare dall'inizio, dalla fine e dalla posizione corrente del file.

11.3 a) `ofPtr = fopen( "oldmast.dat", "r" );`
b) `tfPtr = fopen( "trans.dat", "r" );`
c) `nfPtr = fopen( "newmast.dat", "w" );`
d) `fscanf( ofPtr, "%d%s%f", &accountNum, name, &currentBalance );`
e) `fscanf( tfPtr, "%d%f", &accountNum, &dollarAmount );`
f) `fprintf( nfPtr, "%d%s%.2f", accountNum, name, currentBalance );`

11.4 a) Errore: `"payables.dat"` non è stato aperto prima di usare il suo puntatore di file.
Correzione: usare `fopen` per aprire in scrittura, accodamento o aggiornamento il file `"payables.dat"`.

b) Errore: la funzione `open` non appartiene all'ANSI C.
Correzione: usare la funzione `fopen`.

c) Errore: l'istruzione `fscanf` usa il puntatore di file sbagliato per fare riferimento a `"payables.dat"`.
Correzione: usare il puntatore di file `payPtr` per fare riferimento a `"payables.dat"`.

d) Errore: il contenuto del file sarà eliminato perché è stato aperto in scrittura (`"w"`).
Correzione: per aggiungere dei dati al file, apritelo in aggiornamento (`"r+"`) o in accodamento (`"a"`).

e) Errore: il file `"courses.dat"` è stato aperto per l'aggiornamento con il modo `"w+"` che eliminerà il contenuto corrente del file.
Correzione: aprire il file con il modo `"a"`.

Esercizi

11.5 Riempite gli spazi vuoti in ognuna delle seguenti righe:

a) I computer immagazzinano grandi quantità di dati su dispositivi di memoria secondaria come i _____.

b) Un _____ si compone di molti campi.

c) Un campo che possa contenere numeri, lettere e spazi è un campo _____.

d) Per facilitare il recupero da un file di record specifici, in ognuno di loro un campo sarà scelto come _____.

e) La gran maggioranza delle informazioni immagazzinate nei sistemi di computer è memorizzata in _____ file.

f) Un gruppo di caratteri correlati che trasmettono un'informazione è un _____.

g) I puntatori dei tre file aperti automaticamente quando inizia l'esecuzione di un programma si chiamano _____, _____ e _____.

h) La funzione _____ scrive un carattere in un file specificato.

i) La funzione _____ scrive una riga in un file specificato.

j) La funzione _____ è usata generalmente per scrivere dei dati in un file ad accesso casuale.

k) La funzione _____ riposiziona all'inizio del file il suo puntatore di posizione.

11.6 Determinate quali delle seguenti affermazioni siano *vere* o *false*. Per quelle che ritenete false, spiegatene il motivo.

a) Le impressionanti funzioni eseguite dai computer comportano essenzialmente la manipolazione di zeri e di uno.

b) Le persone preferiscono manipolare i bit, invece dei caratteri o dei campi, perché sono più compatti.

c) Le persone immettono i programmi e le unità di informazione come caratteri; in seguito, i computer li gestiranno e li elaboreranno come gruppi di zero e di uno.

d) Il CAP di una persona è un esempio di campo numerico.

e) Nelle applicazioni per computer, l'indirizzo di una persona è generalmente considerato un campo alfabetico.

f) Le unità di informazione elaborate da un computer formano una gerarchia di dati, nella quale le unità di informazione diventano più grandi e più complesse, man mano che si procede dai campi ai caratteri, ai bit ecc.

g) Una chiave del record identifica quest'ultimo come un'entità appartenente a un particolare campo.

h) Per facilitare le elaborazioni del computer, la maggior parte delle aziende immagazzina tutte le proprie informazioni in un unico file.
i) Nei programmi in C si punta sempre ai file con il loro nome.
j) Ogni volta che un programma crea un file, questo sarà conservato automaticamente dal computer per i riferimenti futuri.

11.7 L'Esercizio 11.3 ha chiesto al lettore di scrivere una serie di istruzioni distinte. In realtà, quelle istruzioni formano il nucleo di un importante tipo di programma per l'elaborazione dei file, ovverosia, un programma di file-matching (confronto di file). Nelle elaborazioni dei dati commerciali, è uso comune avere diversi file in ogni sistema. In un sistema di gestione dei crediti, per esempio, c'è generalmente un file principale che contiene informazioni dettagliate su ogni cliente, come il suo nome, l'indirizzo, il numero di telefono, gli insoluti, il limite di credito, le condizioni di sconto, le disposizioni contrattuali e possibilmente un succinto riepilogo dei suoi recenti acquisti e pagamenti.

Man mano che le transazioni verranno concluse (ovverosia, verranno effettuate delle vendite e arriveranno dei pagamenti nella corrispondenza), queste saranno immesse in un file. Alla fine di ogni periodo commerciale (un mese per alcune aziende, una settimana per altre, un giorno in certi casi) il file delle transazioni (chiamato `"trans.dat"` nell'Esercizio 11.3) sarà confrontato con il file principale (chiamato `"oldmast.dat"` nell'Esercizio 11.3), aggiornando così tutti i record di conto con gli ultimi acquisti e pagamenti. Dopo che i suddetti aggiornamenti saranno stati eseguiti, il file principale sarà riscritto in un nuovo archivio (`"newmast.dat"`) che, alla fine del successivo periodo commerciale, sarà usato per ricominciare il processo di aggiornamento.

I programmi di file-matching devono affrontare alcuni problemi che non hanno riscontro in quelli che gestiscono degli archivi singoli. Per esempio, non sempre sarà possibile riscontrare delle corrispondenze tra i record dei due file. Un cliente presente sul file principale potrebbe non aver effettuato alcun acquisto o pagamento nel periodo commerciale corrente e, di conseguenza, per questo cliente non esisterebbe nessun record nel file delle transazioni. Allo stesso modo, un cliente che abbia effettuato qualche acquisto o pagamento potrebbe essere appena giunto nella comunità, e l'azienda potrebbe non aver avuto l'opportunità di creare per lui un record principale.

Usate le istruzioni scritte nell'Esercizio 11.3 come base per scrivere un programma di file-matching per la contabilità del credito. In ognuno dei file, usate il numero di conto come chiave del record per riscontrare le corrispondenze. Supponete che siano tutti file sequenziali con record immagazzinati in ordine crescente di numero di conto.

Ogni volta che avrete riscontrato una corrispondenza (ovverosia, quando avrete trovato due record con lo stesso numero di conto sul file principale e su quello delle transazioni), dovrete aggiungere al saldo corrente del file principale l'ammontare in dollari rilevato dall'archivio delle transazioni e dovrete scrivere il record di "`newmast.dat`". Considerate che gli acquisti saranno indicati sul file delle transazioni con importi positivi, mentre i pagamenti saranno indicati con un ammontare negativo. Nel caso in cui per un certo conto troviate un record principale, ma nessuna corrispondenza nelle transazioni, dovrete semplicemente scrivere il record principale in "`newmast.dat`". Nel caso in cui troviate un record di transazione, ma nessuna corrispondenza tra quelli principali, dovrete visualizzare il messaggio "`Unmatched transaction record for account number ...`" (completate il messaggio, rilevando il numero di conto dal record di transazione).

11.8 Dopo aver scritto il programma dell'Esercizio 11.7, scrivetene uno semplice che crei alcuni dati di prova per controllare il programma dell'Esercizio 11.7. Usate i seguenti dati di conto:

File principale: numero di conto	Nome	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	-14.22

File delle transazioni: numero di conto	Importo in dollari
100	27.14
300	62.11
400	100.56
900	82.17

11.9 Eseguite il programma dell'Esercizio 11.7 usando i file di prova creati nell'Esercizio 11.8. Usate il programma della Sezione 11.7 per visualizzare un nuovo file principale. Verificate attentamente i risultati.

11.10 È probabile (anzi molto comune) che ci siano diversi record di transazione con la stessa chiave. Ciò potrebbe accadere perché, durante un periodo commerciale, un particolare cliente potrebbe effettuare diversi acquisti e pagamenti. Riscrivete il vostro programma di file-matching per la contabilità del credito scritto nell'Esercizio 11.7, in modo che sia possibile gestire svariati record di transazione con la stessa chiave. Modificate i dati di prova dell'Esercizio 11.8, in modo da includere questi ulteriori record di transazione:

Numero di conto	Importo in dollari
300	83.89
700	80.78
700	1.53

11.11 Scrivete delle istruzioni che eseguano ognuno dei seguenti compiti. Supponete che la struttura

```
struct person {
   char lastName[ 15 ];
   char firstName[ 15 ];
   char age[ 4 ];
};
```

sia già stata definita e che il file sia già stato aperto in scrittura.

a) Inizializzate il file `"nameage.dat"` in modo che ci siano 100 record con `lastName = "unassigned"`, `firstName = ""` e `age = "0"`.
b) Immettete `10` cognomi, nomi ed età e scriveteli nel file.
c) Aggiornate un record e; qualora non vi troviate delle informazioni, informate l'utente con il messaggio "`No info`".
d) Eliminate un record che contenga delle informazioni, inizializzando nuovamente quel particolare record.

11.12 Siete il proprietario di una ferramenta e avete bisogno di mantenere un inventario che possa dirvi quali e quanti attrezzi avete, e il costo di ognuno di essi. Scrivete un programma che inizializzi il file `"hardware.dat"` con 100 record vuoti, vi consenta di immettere i dati relativi a ogni attrezzo, vi dia la possibilità di elencarli tutti, vi lasci eliminare il record di un attrezzo che non avete più e vi permetta di aggiornare qualsiasi informazione all'interno del file. Il numero di identificazione dell'attrezzo dovrà essere anche quello del record. Usate le seguenti informazioni come dati iniziali per il vostro file.

Record n.	Nome dell'attrezzo	Quantità	Costo
3	Smerigliatrice elettrica	7	57.98
17	Martello	76	11.99
24	Sega da traforo	21	11.00
39	Falciatrice	3	79.50
56	Sega elettrica	18	99.99
68	Giravite	106	6.99
77	Martello da fabbro	11	21.50
83	Chiave inglese	34	7.50

11.13 *Generatore di parole con i numeri telefonici.* Le tastiere telefoniche standard dei cellulari contengono le cifre da zero a nove. A ognuno dei numeri da due a nove sono associate tre o quattro lettere, come indicato dalla seguente tabella:

Numero	Lettere
2	A B C
3	D E F
4	G H I
5	J K L
6	M N O
7	P Q R S
8	T U V
9	W X Y Z

Per molte persone la memorizzazione dei numeri telefonici è un'operazione difficile e, per questo motivo, utilizzano la corrispondenza tra le cifre e le lettere per sviluppare parole di sette caratteri che corrispondano ai propri numeri telefonici. Per esempio, una persona il cui numero telefonico sia 468-6374, potrebbe usare le corrispondenze indicate nella tabella precedente per sviluppare la parola di sette lettere "INUMERI".

Le aziende tentano spesso di ottenere dei numeri telefonici che possano essere facilmente ricordati dai loro clienti. Se per le telefonate dei propri clienti un'azienda potesse pubblicizzare una semplice parola, invece di un numero telefonico, allora quell'azienda riceverebbe indubbiamente qualche telefonata in più.

Ogni parola di sette lettere corrisponde esattamente a un numero telefonico di altrettante cifre. Un ristorante che desiderasse incrementare gli introiti dei propri affari, potrebbe sicuramente farlo con il numero 872-4266 (ovverosia "USCIAMO").

A ogni numero telefonico di sette cifre corrispondono molte distinte parole di altrettante lettere. Sfortunatamente, molte di queste rappresentano giustapposizioni di lettere senza significato. Tuttavia, è probabile che il proprietario di una bottega di barbiere possa essere contento di sapere che il numero telefonico del proprio salone, 227-3554, corrisponda a "CAPELLI". Il proprietario di un negozio di alcolici sarebbe senza dubbio felice di scoprire che il numero telefonico del proprio negozio, 547-8674, coincida con "LIQUORI". Un vegetariano con il numero telefonico 837-3872 sarebbe compiaciuto di sapere che quelle cifre equivalgano alla parola "VERDURA".

Scrivete un programma C che, dato un numero di sette cifre, scriva in un file ogni possibile parola di sette lettere corrispondente a quel numero. Esistono al massimo 16384 parole possibili (4 elevato alla settima potenza). Evitate i numeri di telefono con le cifre zero e uno.

11.14 Nel caso in cui abbiate a disposizione un dizionario computerizzato, modificate il programma che avete scritto nell'Esercizio 11.13 in modo che ricerchi le parole nel dizionario. Alcune delle combinazioni di sette lettere create da questo programma consistono di due o più parole (il numero telefonico 646-2276 corrisponde a "MIOCAPO").

11.15 Modificate l'esempio della Figura 8.14, in modo che utilizzi le funzioni `fgetc` e `fputs`, invece di `getchar` e `puts`. Il programma dovrà offrire all'utente la possibilità di scegliere se leggere dallo standard input e scrivere sullo standard output, o se leggere e scrivere in file specificati. Nel caso in cui l'utente scelga la seconda possibilità, consentitegli di immettere i nomi dei file di input e di output.

11.16 Scrivete un programma che usi l'operatore `sizeof` per determinare le dimensioni in byte dei vari tipi di dato sul vostro sistema. Scrivete i risultati nel file `"datasize.dat"`, così che possiate visualizzarli in un secondo momento. Il formato dei risultati nel file dovrà essere come segue:

```
Data type                  Size
char                          1
unsigned char                 1
short int                     2
unsigned short int            2
int                           4
unsigned int                  4
long int                      4
unsigned long int             4
float                         4
double                        8
long double                  16
```

[*Nota*: sul vostro computer, le dimensioni dei tipi di dato potrebbero non corrispondere a quelle elencate qui sopra.]

11.17 Nell'Esercizio 7.19, avete scritto una simulazione software di un computer che usa uno speciale linguaggio macchina, chiamato Linguaggio Macchina Simpletron (LMS). Ogni volta che volete eseguire un programma LMS, dovete digitare da tastiera il codice per immetterlo nel simulatore. Se vi capitasse di commettere un errore durante la digitazione del programma LMS, dovreste far ripartire il simulatore e digitare nuovamente il codice LMS. Non sarebbe bello poter leggere il programma LMS da un file, piuttosto che digitarlo ogni volta? Ciò ridurrebbe il tempo e gli errori durante la fase di preparazione per l'esecuzione di un programma LMS.

a) Modificate il simulatore che avete scritto nell'Esercizio 7.19, in modo che legga i programmi LMS da un file specificato tramite tastiera dall'utente.

b) Al termine della sua esecuzione, il Simpletron avrà visualizzato sullo schermo il contenuto dei propri registri e della memoria. Sarebbe sicuramente bello poter catturare l'output in un file, perciò modificate il simulatore in modo che scriva il suo output in un file, oltre che sullo schermo.

Capitolo 12

Le strutture di dati in C

Obiettivi

- Essere in grado di allocare e liberare la memoria per gli oggetti di dati in modo dinamico.
- Essere in grado di formare strutture di dati concatenate usando i puntatori, le strutture ricorsive e la ricorsione.
- Essere in grado di creare e gestire le liste concatenate, le code, le pile e gli alberi binari.
- Capire varie importanti applicazioni delle strutture di dati concatenate.

12.1 Introduzione

Abbiamo studiato *strutture di dati* con dimensione fissa, come i vettori a una o a due dimensioni, e le `struct`. Questo capitolo introdurrà le *strutture di dati dinamiche* con dimensioni che crescono e decrescono durante l'esecuzione del programma. Le *liste concatenate* sono collezioni di unità di informazione "messe in fila indiana", nelle quali le inserzioni e le eliminazioni possono essere eseguite in una qualsiasi posizione della lista. Le *pile* (o stack), nelle quali le inserzioni e le eliminazioni sono eseguite solo da una loro estremità, la testa della pila, sono molto importanti per i compilatori e i sistemi operativi. Le *code* rappresentano linee di attesa; le inserzioni sono eseguite all'estremità posteriore delle code (detta anche *fine della coda*), mentre le eliminazioni sono effettuate all'inizio delle stesse (detto anche *testa della coda*). Gli *alberi binari* facilitano le ricerche e gli ordinamenti dei dati ad alta velocità, l'eliminazione efficiente delle unità di informazione duplicate, la rappresentazione di directory del file system e la compilazione delle espressioni in linguaggio macchina. Per ognuna di queste strutture di dati ci sono molte altre interessanti applicazioni.

Discuteremo ognuno dei tipi principali delle suddette strutture di dati e implementeremo dei programmi che le creeranno e le gestiranno. Questo è un capitolo impegnativo. I programmi sono corposi e incorporano la maggior parte di quello che avete appreso nei capitoli precedenti. I programmi fanno un uso pesante della gestione dei puntatori, un argomento che molti considerano essere tra i più difficoltosi del C. Il capitolo è pieno di programmi di elevata utilità che voi sarete in grado di utilizzare nei corsi più avanzati; esso include anche una ricca collezione di esercizi che evidenziano le applicazioni pratiche delle strutture di dati.

Noi speriamo sinceramente che vogliate provare a realizzare il progetto più importante, descritto nella sezione speciale "Costruire il vostro compilatore". Sinora avete usato un com-

pilatore per tradurre in linguaggio macchina i vostri programmi C, in modo da poterli eseguire sul vostro computer. In questo progetto, costruirete realmente il vostro compilatore. Questo leggerà un file di istruzioni scritte in un semplice ma potente linguaggio di alto livello, simile alle prime versioni del popolare linguaggio BASIC. Il vostro compilatore tradurrà le suddette istruzioni in un file di codice scritto in Linguaggio Macchina Simpletron. LMS è il linguaggio che avete appreso nel Capitolo 7, Sezione speciale: costruite il vostro computer. Il vostro programma di simulazione del Simpletron, infine, eseguirà il codice LMS prodotto dal vostro compilatore! Questo progetto vi darà una meravigliosa opportunità di esercitarvi sulla maggior parte delle cose che avete appreso in questo corso. La sezione speciale vi accompagnerà attentamente attraverso la progettazione del linguaggio di alto livello, descrivendo gli algoritmi di cui avrete bisogno, per convertire ogni tipo di istruzione di alto livello nelle corrispondenti in linguaggio macchina. Nel caso vi piacciano le sfide, potrete tentare di apportare i tanti miglioramenti, per il compilatore e per il simulatore Simpletron, suggeriti nella sezione Esercizi.

12.2 Le strutture ricorsive

Una *struttura ricorsiva* contiene un membro di tipo puntatore, che fa riferimento a una struttura dello stesso tipo di quella in cui è contenuto. Per esempio, la definizione

```
struct node {
   int data;
   struct node *nextPtr;
};
```

definisce il tipo `struct node`. Una struttura di tipo `struct node` ha due membri, l'intero `data` e il puntatore `nextPtr`. Il membro `nextPtr` punta a `struct node`, ossia a una struttura dello stesso tipo di quella che stiamo dichiarando in questo momento, da ciò il termine "*struttura ricorsiva*". Il membro `nextPtr` è detto *link* (legame): in altre parole, `nextPtr` può essere usato per "legare o concatenare" una struttura `struct node` a un'altra dello stesso tipo. Le strutture ricorsive possono essere legate insieme per formare utili organizzazioni di dati come le liste, le code, le pile e gli alberi. La Figura 12.1 illustra due strutture ricorsive che sono state concatenate per formare una lista. Notate che, nel membro di legame della seconda struttura ricorsiva, è stata posta una linea obliqua, che rappresenta un puntatore `NULL`, per indicare che il legame non fa riferimento a un'altra struttura. [Nota: La linea obliqua ha solo uno scopo illustrativo; non corrisponde al carattere backslash (`"\"`) del C.] Un puntatore `NULL` normalmente indica la fine della struttura di dati, proprio come il carattere nullo indica la fine di una stringa.

Errore tipico 12.1

Dimenticare di impostare a `NULL` *il link dell'ultimo nodo di una lista può provocare degli errori a tempo di esecuzione.*

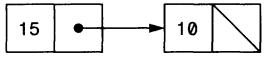


Figura 12.1 Due strutture ricorsive concatenate

12.3 Allocazione dinamica della memoria

Creare e gestire strutture di dati dinamiche richiede l'*allocazione dinamica della memoria*: in altre parole, la capacità di un programma di ottenere, durante la sua esecuzione, un maggior spazio di memoria per immagazzinare i nuovi nodi e di poterlo rilasciare quando non sarà più necessario. Il limite per l'allocazione dinamica della memoria potrebbe essere grande quanto la quantità di memoria fisica disponibile nel computer, o quanto l'ammontare di quella disponibile in un sistema a memoria virtuale. I limiti sono spesso più piccoli, perché la memoria disponibile deve essere condivisa tra molte applicazioni.

Le funzioni `malloc`, `free` e l'operatore `sizeof` sono essenziali per l'allocazione dinamica della memoria. La funzione `malloc` acquisisce, come argomento, il numero di byte che dovranno essere allocati e restituisce un puntatore di tipo `void *` (puntatore a `void`), che fa riferimento all'area di memoria allocata. Un puntatore `void *` può essere assegnato a una variabile di uno qualsiasi degli altri tipi di puntatore. La funzione `malloc` è normalmente usata con l'operatore `sizeof`. Per esempio, l'istruzione

```
newPtr = malloc( sizeof( struct node ) );
```

valuterà `sizeof( struct node )`, per determinare la dimensione in byte della struttura di tipo `struct node`, allocherà una nuova area nella memoria, costituita da quel numero di byte, e immagazzinerà nella variabile `newPtr` un puntatore alla memoria allocata. La memoria allocata non viene inizializzata. Qualora non vi fosse memoria disponibile, `malloc` restituirebbe `NULL`.

La funzione `free` rilascia la memoria: in altre parole, questa è restituita al sistema in modo che in futuro possa essere allocata nuovamente. Per liberare la memoria allocata dinamicamente dalla precedente chiamata a `malloc`, dovrete usare l'istruzione

```
free(newPtr);
```

Le prossime sezioni discuteranno delle liste, delle pile, delle code e degli alberi, ognuno dei quali sarà creato e gestito con un'allocazione dinamica della memoria e con le strutture ricorsive.

Obiettivo portabilità 12.1

La dimensione di una struttura non è necessariamente la somma di quelle dei suoi membri. Ciò a causa delle esigenze di allineamento dipendenti dalle diverse macchine (consultate il Capitolo 10).

Errore tipico 12.2

Presumere che la dimensione di una struttura sia semplicemente la somma di quelle dei suoi membri è un errore logico.

Buona abitudine 12.1

Usate l'operatore ***sizeof*** *per determinare la dimensione di una struttura.*

Collaudo e messa a punto 12.1

Quando utilizzate `malloc`*, ricordatevi di controllare se il valore restituito è un puntatore* `NULL`*. Visualizzate un messaggio di errore qualora la memoria richiesta non possa essere allocata.*

Errore tipico 12.3

Non rilasciare la memoria allocata dinamicamente, quando non è più necessaria, potrebbe causare un esaurimento prematuro della memoria del sistema. Questo tipo di errore a volte è chiamato "memory leak" (perdita o dispersione di memoria).

Buona abitudine 12.2

Nel momento in cui la memoria che avevate allocato dinamicamente non è più necessaria, usate `free` *per rilasciarla immediatamente al sistema.*

Errore tipico 12.4

Rilasciare con ***free*** *la memoria che non è stata allocata dinamicamente con* `malloc` *è un errore.*

Errore tipico 12.5

Puntare a un'area di memoria che è stata rilasciata con `free` *è un errore.*

12.4 Le liste concatenate

Una *lista concatenata* è una collezione lineare di strutture ricorsive (*nodi*) connesse da puntatori, detti *link* (anelli o collegamenti), da cui il termine lista "concatenata". Vi si accede per mezzo di un puntatore al suo primo nodo. A quelli successivi, si accede per mezzo del puntatore di concatenamento immagazzinato in ogni nodo. Per convenzione, il puntatore di concatenamento dell'ultimo nodo inserito in una lista è impostato a `NULL`, in modo da marcare la fine della stessa. In una lista concatenata i dati sono immagazzinati in modo dinamico: ossia, ogni nodo sarà creato solo quando sarà necessario. Un nodo può contenere dati di ogni tipo, incluse altre `struct`. Anche le pile e le code sono delle strutture lineari di dati e, come vedremo, sono versioni limitate delle liste concatenate. Gli alberi invece sono strutture di dati non lineari.

Le liste di dati potrebbero essere immagazzinate in vettori, ma le liste concatenate offrono diversi vantaggi. Una lista concatenata sarà appropriata quando non sarà possibile determinare, a priori, il numero delle unità di informazione che dovranno essere rappresentate nella struttura di dati. Le liste concatenate sono dinamiche, perciò la loro lunghezza potrà crescere o decrescere secondo necessità. La dimensione di un vettore, invece, non può essere alterata, dopo che la memoria è stata allocata. I vettori possono riempirsi. Le liste concatenate, invece, si riempiono solo quando il sistema non ha più memoria sufficiente per soddisfare le richieste di allocazione.

Obiettivo efficienza 12.1

Un vettore potrebbe essere dichiarato in modo da contenere più elementi di quanti fossero stati previsti, ma questo potrebbe costituire uno spreco di memoria. In queste situazioni, le liste concatenate possono offrire un miglior utilizzo della memoria.

Le liste concatenate possono essere mantenute in modo ordinato, inserendo ogni nuovo elemento nella posizione appropriata all'interno della lista.

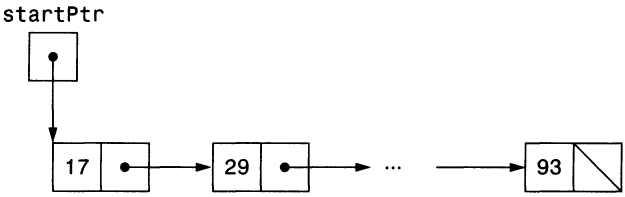


Figura 12.2 Una rappresentazione grafica di una lista concatenata

Obiettivo efficienza 12.2

In un vettore ordinato, l'inserimento e l'eliminazione potrebbero richiedere del tempo: si dovrà far scorrere, in modo appropriato, tutti gli elementi successivi a quello inserito o rimosso.

Obiettivo efficienza 12.3

Gli elementi di un vettore sono immagazzinati nella memoria in modo contiguo. Ciò consentirà un accesso immediato a ognuno dei suoi elementi, perché l'indirizzo di questi potrà essere calcolato direttamente, in base alla loro posizione rispetto all'inizio del vettore. Le liste concatenate non permettono un accesso così immediato ai propri elementi.

I nodi delle liste concatenate normalmente non sono immagazzinati in memoria in modo contiguo. Dal punto di vista logico, tuttavia, i nodi di una lista concatenata sembrano essere contigui. La Figura 12.2 mostra una lista concatenata con diversi nodi.

Obiettivo efficienza 12.4

Usare l'allocazione dinamica della memoria (invece dei vettori) per le strutture di dati che crescono e decrescono durante l'esecuzione del programma, potrebbe far risparmiare la memoria. Tenete a mente, però, che i puntatori sottraggono spazio e che l'allocazione dinamica della memoria incorre nell'onere causato dalle chiamate di funzione.

Il programma della Figura 12.3 (l'output è mostrato nella Figura 12.4) gestirà una lista di caratteri. Il programma offrirà due scelte: 1) inserire un carattere nella lista in ordine alfabetico (funzione `insert`), e 2) eliminare un carattere dalla lista (funzione `delete`). Questo è un programma lungo e complesso che sarà discusso in dettaglio. L'Esercizio 12.20 chiederà al lettore di implementare una funzione ricorsiva che visualizzi una lista in ordine inverso. L'Esercizio 12.21 chiederà al lettore di implementare una funzione ricorsiva che cerchi in una lista concatenata una particolare unità di informazione.

```
1  /* Fig. 12.3: fig12_03.c
2     Manipolazione e manutenzione di una lista */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* struttura ricorsiva */
7  struct listNode {
8     char data; /* ogni listNode contiene un carattere */
```

Figura 12.3 Inserire ed eliminare i nodi in una lista (continua)

```
9     struct listNode *nextPtr; /* puntatore al prossimo nodo */
10 }; /* fine della struttura listNode */
11
12 typedef struct listNode ListNode; /* sinonimo di struct listNode */
13 typedef ListNode *ListNodePtr; /* sinonimo di ListNode* */
14
15 /* prototitpi */
16 void insert( ListNodePtr*sPtr, char value );
17 char delete( ListNodePtr*sPtr, char value );
18 int isEmpty( ListNodePtr sPtr );
19 void printList( ListNodePtr currentPtr );
20 void instructions( void );
21
22 int main()
23 {
24    ListNodePtrstartPtr = NULL; /* all'inizio non ci sono nodi */
25    int choice; /* scelta dell'utente */
26    char item; /* carattere digitato dall'utente */
27
28    instructions();  /* visualizza il menu */
29    printf( "? " );
30    scanf( "%d", &choice );
31
32    /* esegue il ciclo finché l'utente non sceglie 3 */
33    while ( choice != 3 ) {
34
35       switch ( choice ) {
36
37          case 1:
38             printf( "Enter a character: " );
39             scanf( "\n%c", &item );
40             insert( &startPtr, item ); /* inserisce l'elemento
                                             nella lista */
41             printList( startPtr );
42             break;
43
44          case 2:
45
46             /* se la lista non è vuota */
47             if ( !isEmpty( startPtr ) ) {
48                printf( "Enter character to be deleted: " );
49                scanf( "\n%c", &item );
50
51                /* se il carattere viene trovato, lo si rimuove */
52                if ( delete( &startPtr, item ) ) {
53                   printf( "%c deleted.\n", item );
54                   printList( startPtr );
55                } /* fine del ramo if */
56                else {
```

Figura 12.3 Inserire ed eliminare i nodi in una lista (continua)

```
57                      printf( "%c not found.\n\n", item );
58                   } /* fine del ramo else */
59
60                } /* fine del ramo if */
61                else {
62                   printf( "List is empty.\n\n" );
63                }
64
65                break;
66
67             default:
68                printf( "Invalid choice.\n\n" );
69                instructions();
70                break;
71
72          } /* fine del comando switch */
73
74          printf( "? " );
75          scanf( "%d", &choice );
76       } /* fine del comando while */
77
78       printf( "End of run.\n" );
79
80       return 0; /* indica che il programma è terminato con successo */
81
82    } /* fine della funzione main */
83
84    /* Visualizza le istruzioni del programma all'utente */
85    void instructions( void )
86    {
87       printf( "Enter your choice:\n"
88               "   1 to insert an element into the list.\n"
89               "   2 to delete an element from the list.\n"
90               "   3 to end.\n" );
91    } /* fine della funzione instructions */
92
93    /* Inserisce ordinatamente un nuovo valore nella lista */
94    void insert( ListNodePtr *sPtr, char value )
95    {
96       ListNodePtr newPtr; /* puntatore al nuovo nodo */
97       ListNodePtr previousPtr; /* puntatore al nodo precedente
                                     nella lista */
98       ListNodePtr currentPtr; /* puntatore al nodo corrente nella lista */
99
100      newPtr = malloc( sizeof( ListNode ) ); /* crea il nodo */
101
102      if ( newPtr != NULL ) {    /* la memoria è disponibile? */
103         newPtr->data = value; /* memorizza il valore nel nodo */
```

Figura 12.3 Inserire ed eliminare i nodi in una lista (continua)

```
104         newPtr->nextPtr = NULL; /* il nodo non è collegato
                                       a un altro nodo */
105
106         previousPtr = NULL;
107         currentPtr = *sPtr;
108
109         /* esegue un ciclo per trovare la posizione corretta
               nella lista */
110         while ( currentPtr != NULL && value > currentPtr->data ) {
111            previousPtr = currentPtr;          /* passa al...   */
112            currentPtr = currentPtr->nextPtr;  /* ... prossimo nodo */
113         } /* fine del comando while */
114
115         /* inserisce il nuovo nodo all'inizio della lista */
116         if ( previousPtr == NULL ) {
117            newPtr->nextPtr = *sPtr;
118            *sPtr = newPtr;
119         } /* fine del ramo if */
120         else { /* inserisce il nuovo nodo tra previousPtr e currentPtr */
121            previousPtr->nextPtr = newPtr;
122            newPtr->nextPtr = currentPtr;
123         } /* fine del ramo else */
124
125      } /* fine del ramo if */
126      else {
127         printf( "%c not inserted. No memory available.\n", value );
128      } /* fine del ramo else */
129
130   } /* fine della funzione insert */
131
132   /* Elimina un elemento della lista */
133   char delete( ListNodePtr *sPtr, char value )
134   {
135      ListNodePtr previousPtr; /* puntatore al nodo precedente
                                    nella lista */
136      ListNodePtr currentPtr; /* puntatore al nodo corrente
                                   nella lista */
137      ListNodePtr tempPtr; /* puntatore a un nodo temporaneo */
138
139      /* elimina il primo nodo */
140      if ( value == ( *sPtr )->data ) {
141         tempPtr = *sPtr; /* memorizza il nodo che sta per essere
                               rimosso */
142         *sPtr = ( *sPtr )->nextPtr;  /* "sfila" il nodo */
143         free( tempPtr );   /* libera la memoria usata dal nodo "sfilato" */
144         return value;
145      } /* fine del ramo if */
146      else {
147         previousPtr = *sPtr;
148         currentPtr = ( *sPtr )->nextPtr;
```

Figura 12.3 Inserire ed eliminare i nodi in una lista (continua)

```
149
150         /* esegue un ciclo per trovare la posizione corretta
               nella lista */
151         while ( currentPtr != NULL && currentPtr->data != value ) {
152            previousPtr = currentPtr;          /* passa al...   */
153            currentPtr = currentPtr->nextPtr;  /* ... prossimo nodo */
154         } /* fine del comando while */
155
156         /* elimina il nodo puntato da currentPtr */
157         if ( currentPtr != NULL ) {
158            tempPtr = currentPtr;
159            previousPtr->nextPtr = currentPtr->nextPtr;
160            free( tempPtr );
161            return value;
162         } /* fine del comando if */
163
164      } /* fine del ramo else */
165
166      return '\0';
167
168   } /* fine della funzione delete */
169
170   /* Se la lista è vuota, restituisce 1 altrimenti 0 */
171   int isEmpty( ListNodePtr sPtr )
172   {
173      return sPtr == NULL;
174
175   } /* fine della funzione isEmpty */
176
177   /* Visualizza la lista */
178   void printList( ListNodePtr currentPtr )
179   {
180
181      /* se la lista è vuota */
182      if ( currentPtr == NULL ) {
183         printf( "List is empty.\n\n" );
184      }
185      else {
186         printf( "The list is:\n" );
187
188         /* finché non raggiunge la fine della lista */
189         while ( currentPtr != NULL ) {
190            printf( "%c --> ", currentPtr->data );
191            currentPtr = currentPtr->nextPtr;
192         } /* fine del comando while */
193
194         printf( "NULL\n\n" );
195      } /* fine del ramo else */
196
197   } /* fine della funzione printList */
```

Figura 12.3 Inserire ed eliminare i nodi in una lista

Le funzioni principali delle liste concatenate sono `insert` (righe 94-130) e `delete` (righe 133-168). La funzione `isEmpty` (righe 171-175) è detta *funzione predicato*, perché non altera la lista in alcun modo, ma determinerà piuttosto se sia vuota (ovverosia, se il puntatore al primo nodo della lista sia un `NULL`). In questo caso la funzione restituirebbe il valore `1`, invece di `0`. La funzione `printList` (righe 178-197) visualizzerà la lista.

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.
```

Figura 12.4 L'output di esempio per il programma della Figura 12.3 (continua)

```
? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Figura 12.4 L'output di esempio per il programma della Figura 12.3

I caratteri saranno inseriti nella lista in ordine alfabetico. La funzione `insert` (righe 94-130) riceverà l'*indirizzo* della lista e il carattere che dovrà essere inserito. L'indirizzo della lista è necessario qualora il valore debba essere inserito all'inizio della stessa. Fornire l'indirizzo della lista permette alla stessa (ovverosia, al puntatore al suo primo nodo) di essere modificata attraverso una chiamata per riferimento. Dato che la lista stessa è un puntatore (al suo primo elemento), passare l'indirizzo della lista crea un *puntatore a puntatore* (in altre parole, una *doppia indirezione*). Questa è una nozione complessa che richiede una programmazione attenta. I passi per inserire un carattere all'interno della lista sono i seguenti (consultate la Figura 12.5):

1. Creare un nodo richiamando `malloc` e assegnare l'indirizzo della memoria allocata a `newPtr` (riga 100), il carattere da inserire a `newPtr->data` (riga 103) e `NULL` a `newPtr->nextPtr` (riga 104).
2. Inizializzare `previousPtr` a `NULL` (riga 106) e `currentPtr` a `*sPtr` (riga 107), il puntatore all'inizio della lista. I puntatori `previousPtr` e `currentPtr` immagazzineranno, rispettivamente, le locazioni del nodo precedente e di quello successivo al punto di inserimento.
3. Fintanto che `currentPtr` sarà diverso da `NULL` e il valore da inserire sarà maggiore di `currentPtr->data` (riga 110), assegnare `currentPtr` a `previousPtr` (riga 111) e fare avanzare `currentPtr` al nodo successivo della lista (riga 112). Questo individuerà il punto di inserimento del valore.
4. Nel caso in cui `previousPtr` sia `NULL` (riga 116), il nuovo nodo dovrà essere inserito come primo elemento della lista (righe 117-118). Assegnare `*sPtr` a `newPtr->nextPtr` (il nuovo link punterà così all'ex primo nodo), e `newPtr` a `*sPtr` (`*sPtr` punterà così al nuovo nodo). Altrimenti, nel caso in cui `previousPtr` non sia `NULL`, il nuovo nodo dovrà essere inserito al posto giusto (righe 121-122). Assegnare `newPtr` a `previousPtr->nextPtr` (il nodo precedente punterà così a quello nuovo), e `currentPtr` a `newPtr->nextPtr` (il link del nuovo nodo punterà così a quello corrente).

Collaudo e messa a punto 12.2

Assegnate `NULL` *al membro link di un nuovo nodo. Prima di essere utilizzati, i puntatori devono sempre essere inizializzati.*

La Figura 12.5 mostra l'inserimento in una lista ordinata di un nodo che contiene il carattere `'C'`. La parte a) della figura mostra la lista e il nuovo nodo, prima dell'inserimento.

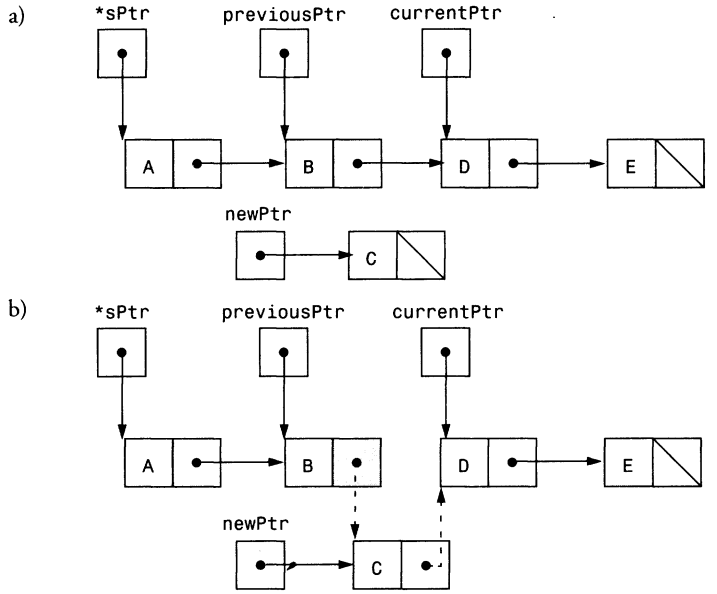


Figura 12.5 Inserire ordinatamente un nodo in una lista

La parte b) della figura mostra il risultato dell'inserimento del nuovo nodo. I puntatori aggiornati sono rappresentati dalle linee tratteggiate.

La funzione `delete` (righe 137-168) riceverà l'indirizzo del puntatore all'inizio della lista e il carattere che dovrà essere eliminato. I passi per l'eliminazione di un carattere dalla lista sono i seguenti:

1) Nel caso in cui il carattere da eliminare corrisponda a quello contenuto nel primo nodo della lista (riga 140), assegnare `*sPtr` a `tempPtr` (`tempPtr` sarà usato per rilasciare con `free` la memoria non più necessaria), `(*sPtr)->nextPtr` a `*sPtr` (`*sPtr` punterà così al secondo nodo della lista), rilasciare con `free` la memoria puntata da `tempPtr` e restituire il carattere eliminato.
2) Altrimenti, inizializzare `previousPtr` con `*sPtr` e `currentPtr` con `(*sPtr)->nextPtr` (righe 147-148).
3) Fintanto che `currentPtr` non sia `NULL` e il valore da eliminare non sia uguale a `currentPtr->data` (riga 151), assegnare `currentPtr` a `previousPtr` (riga 152) e `currentPtr->nextPtr` a `currentPtr` (riga 153). Questo determinerà se il carattere da eliminare sia contenuto nella lista.
4) Nel caso in cui `currentPtr` non sia `NULL` (riga 157), assegnare `currentPtr` a `tempPtr` (riga 158), `currentPtr->nextPtr` a `previousPtr->nextPtr` (riga 159), rilasciare la memoria del nodo puntato da `tempPtr` (riga 160) e restituire il carattere eliminato dalla lista (riga 161). Nel caso in cui `currentPtr` sia `NULL`, restituire il carattere `NULL` (`'\0'`) per indicare che quello da eliminare non è stato ritrovato nella lista (riga 166).

La Figura 12.6 mostra l'eliminazione di un nodo da una lista concatenata. La parte a) della figura mostra la lista concatenata dopo la precedente operazione di inserimento. La parte b) mostra la nuova assegnazione dell'elemento link di `previousPtr` e l'assegnazione di

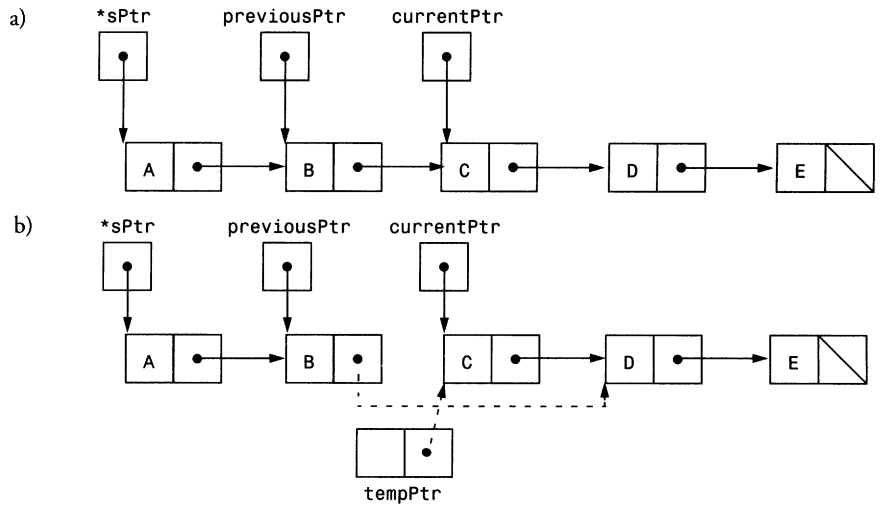


Figura 12.6 Eliminare un nodo da una lista

`currentPtr` a `tempPtr`. Il puntatore `tempPtr` sarà usato per rilasciare la memoria allocata per immagazzinare `'C'`.

La funzione `printList` (righe 178-197) riceverà come argomento un puntatore all'inizio della lista, chiamandolo `currentPtr`. La funzione determinerà in primo luogo se la lista è vuota (righe 182-184) e, in questo caso, `printList` visualizzerà il messaggio `"The list is empty."` e terminerà la propria esecuzione. In caso contrario, visualizzerà i dati contenuti nella lista (righe 185-195). Fintanto che `currentPtr` non sia `NULL`, `currentPtr->data` sarà visualizzato dalla funzione e `currentPtr->nextPtr` sarà assegnato a `currentPtr`. Osservate che qualora il link nell'ultimo nodo della lista non sia `NULL`, l'algoritmo della funzione tenterà di visualizzare dei dati inesistenti, cercandoli oltre la fine della lista, e ciò provocherà un errore. L'algoritmo di visualizzazione per le liste concatenate, le pile e le code è identico.

12.5 Le pile

Una *pila* (o stack) è una versione limitata di una lista concatenata. I nuovi nodi possono essere aggiunti alla pila e rimossi dalla stessa soltanto dalla sua testa. Per questo motivo, una pila è detta anche struttura di dati *last-in, first-out* (*LIFO*, l'ultimo a entrare è il primo a uscire). Si fa riferimento a una pila mediante un puntatore all'elemento in cima alla stessa. Il membro link nell'ultimo nodo della pila è impostato a `NULL`, per indicare la fine della stessa.

La Figura 12.7 mostra una pila con diversi nodi. Notate che le pile e le liste concatenate sono rappresentate in modo identico. La differenza tra le pile e le liste concatenate è che in queste il punto di inserimento e di eliminazione potrebbe essere ovunque, mentre in una pila corrisponde sempre alla sua testa.

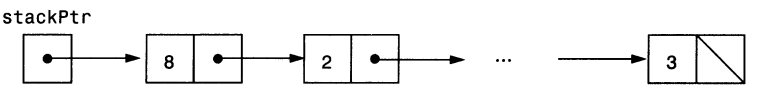


Figura 12.7 Una rappresentazione grafica di una pila

Errore tipico 12.6

Non impostare a `NULL` *il membro link nell'ultimo nodo di una pila può provocare degli errori a tempo di esecuzione.*

Le principali funzioni usate per gestire una pila sono `push` e `pop`. La funzione `push` crea un nuovo nodo e lo posiziona in cima alla pila. La funzione `pop` rimuove un nodo dalla testa della pila, rilascia la memoria allocata dal nodo estratto e restituisce il valore che conteneva.

Il programma della Figura 12.8 (l'output è mostrato nella Figura 12.9) implementerà una semplice pila di valori interi. Il programma offrirà tre scelte: 1) inserire un valore nella pila (funzione `push`), 2) estrarre un valore dalla pila (funzione `pop`), e 3) terminare l'esecuzione del programma.

La funzione `push` (righe 84-100) inserirà un nuovo nodo in cima allo stack; i passi necessari per compiere tale operazione sono i seguenti:

1) Creare un nuovo nodo richiamando la funzione `malloc` e assegnare l'indirizzo della memoria allocata a `newPtr` (riga 88).
2) Assegnare a `newPtr->data` il valore da inserire nella pila (riga 92) e assegnare `*topPtr` (il puntatore alla testa della pila) a `newPtr->nextPtr` (riga 93): il membro link in `newPtr` punterà così al nodo che, in precedenza, occupava la cima della pila.
3) Assegnare `newPtr` a `*topPtr` (riga 94): `*topPtr` punterà così alla nuova testa della pila.

```
1  /* Fig. 12.8: fig12_08.c
2     programma per una pila dinamica */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* struttura ricorsiva */
7  struct stackNode {
8     int data; /* definisce data come un elemento di tipo int */
9     struct stackNode *nextPtr; /* puntatore a elementi di tipo
                                    stackNode */
10 }; /* fine della struttura stackNode */
11
12 typedef struct stackNode StackNode; /* sinonimo di struct stackNode */
13 typedef StackNode *StackNodePtr; /* sinonimo di StackNode* */
14
15 /* prototipi */
16 void push( StackNodePtr *topPtr, int info );
17 int pop( StackNodePtr *topPtr );
18 int isEmpty( StackNodePtr topPtr );
19 void printStack( StackNodePtr currentPtr );
20 void instructions( void );
21
22 int main()
24 {
25    StackNodePtr stackPtr = NULL;  /* punta alla testa della pila */
26    int choice; /* scelta del menu da parte dell'utente */
```

Figura 12.8 Un semplice programma per la gestione di una pila (continua)

```
27      int value; /* elemento di tipo int inserito dall'utente */
28
29      instructions(); /* visualizza il menu */
30      printf( "? " );
31      scanf( "%d", &choice );
32
33      /* esegue il ciclo finché l'utente non sceglie 3 */
34      while ( choice != 3 ) {
35
36         switch ( choice ) {
37
38            /* aggiunge un valore in cima alla pila */
39            case 1:
40               printf( "Enter an integer: " );
41               scanf( "%d", &value );
42               push( &stackPtr, value );
43               printStack( stackPtr );
44               break;
45
46            /* estrae un valore dalla pila */
47            case 2:
48
49               /* se la pila non è vuota */
49               if ( !isEmpty( stackPtr ) ) {
50                  printf( "The popped value is %d.\n", pop( &stackPtr ) );
52               } /* fine del comando if */
53
54               printStack( stackPtr );
55               break;
56
57            default:
58               printf( "Invalid choice.\n\n" );
59               instructions();
60               break;
61
62         } /* fine del comando switch */
63
64         printf( "? " );
65         scanf( "%d", &choice );
66      } /* fine del comando while */
67
68      printf( "End of run.\n" );
69
70      return 0; /* indica che il programma è terminato con successo */
71
72   } /* fine della funzione main */
73
74   /* Visualizza le istruzioni del programma all'utente */
75   void instructions( void )
```

Figura 12.8 Un semplice programma per la gestione di una pila (continua)

```
{
   printf( "Enter choice:\n"
           "1 to push a value on the stack\n"
           "2 to pop a value off the stack\n"
           "3 to end program\n" );
} /* fine della funzione instructions */

/* Inserisce un nodo in cima alla pila */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr; /* puntatore al nuovo nodo */

   newPtr = malloc( sizeof( StackNode ) );

   /* inserisce il nodo in cima alla pila */
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   } /* fine del ramo if */
   else { /* memoria non disponibile */
      printf( "%d not inserted. No memory available.\n", info );
   } /* fine del ramo else */

} /* fine della funzione push */

/* Rimuove un nodo dalla cima della pila */
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr; /* puntatore a un nodo temporaneo */
   int popValue; /* valore del nodo */

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );

   return popValue;

} /* fine della funzione pop */

/* Visualizza la pila */
void printStack( StackNodePtr currentPtr )
{

   /* se la pila non è vuota */
   if ( currentPtr == NULL ) {
      printf( "The stack is empty.\n\n" );
   } /* fine del ramo if */
   else {
```

Figura 12.8 Un semplice programma per la gestione di una pila (continua)

```
126         printf( "The stack is:\n" );
127
128         /* finché non raggiunge la fine della pila */
129         while ( currentPtr != NULL ) {
130            printf( "%d --> ", currentPtr->data );
131            currentPtr = currentPtr->nextPtr;
132         } /* fine del comando while */
133
134         printf( "NULL\n\n" );
135      } /* fine del ramo else */
136
137   } /* fine della funzione printStack */
138
139   /* Se la pila è vuota, restituisce 1 altrimenti 0 */
140   int isEmpty( StackNodePtr topPtr )
141   {
142      return topPtr == NULL;
143
144   } /* fine della funzione isEmpty */
```

Figura 12.8 Un semplice programma per la gestione di una pila

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL
? 1
Enter an integer: 6
The stack is:
6--> 5 -- NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL

? 2
The popped value is 6.
The stack is:
5 --> NULL
```

Figura 12.9 L'output di esempio per il programma della Figura 12.8 (continua)

```
? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Figura 12.9 L'output di esempio per il programma della Figura 12.8

Le manipolazioni che interesseranno `*topPtr` cambieranno il valore di `stackPtr` all'interno della funzione `main`. La Figura 12.10 mostra la funzione `push`. La parte a) della figura mostra la pila e il nuovo nodo, prima dell'intervento di `push`. Le linee tratteggiate nella parte b) illustrano i passi 2 e 3 dell'operazione di `push`, che consentiranno al nodo contenente 12 di diventare la nuova testa della pila.

La funzione `pop` (righe 103-115) rimuoverà un nodo dalla cima della pila. Osservate che prima di richiamare `pop`, la funzione `main` determinerà se la pila è vuota. L'operazione di `pop` consiste di cinque passi:

1) Assegnare `*topPtr` a `tempPtr` (riga 108); `tempPtr` sarà usato per rilasciare la memoria non più necessaria.
2) Assegnare `(*topPtr)->data` a `popValue` (riga 109) per salvare il valore immagazzinato nel nodo di testa.

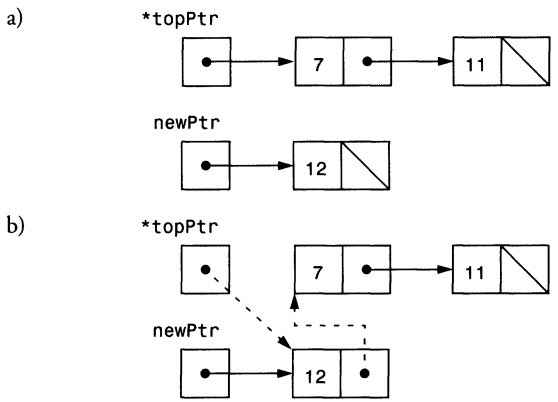


Figura 12.10 L'operazione di push

3) Assegnare `(*topPtr)->nextPtr` a `*topPtr` (riga 110) in modo che `*topPtr` contenga l'indirizzo del nuovo nodo di testa.
4) Rilasciare la memoria puntata da `tempPtr` (riga 111).
5) Restituire `popValue` alla funzione chiamante (riga 113).

La Figura 12.11 mostra la funzione `pop`. La parte a) mostra la pila, dopo la precedente operazione di `push`. La parte b) mostra `tempPtr` mentre punta al primo nodo della pila, mentre `topPtr` punta al secondo. La funzione `free` sarà usata per rilasciare la memoria puntata da `tempPtr`.

Le pile hanno molte interessanti applicazioni. Per esempio, ogni volta che si richiama una funzione, questa deve sapere come poter restituire il controllo alla funzione chiamante, perciò l'indirizzo di ritorno dovrà essere depositato in una pila. Qualora si verifichino molte chiamate di funzione, i diversi valori di ritorno saranno depositati in una pila, secondo l'ordine last-in first-out, in modo che ogni funzione possa restituire il controllo al proprio chiamante. Le pile supportano le chiamate di funzione ricorsive, allo stesso modo di quelle convenzionali non ricorsive.

Le pile contengono anche lo spazio creato per le variabili automatiche di ogni invocazione di funzione. Questo spazio sarà rimosso dalla pila e le suddette variabili non saranno più visibili all'interno del programma, nel momento in cui la funzione restituirà il controllo al suo chiamante.

Le pile sono usate anche dai compilatori, durante il processo di valutazione delle espressioni e quello di generazione del codice in linguaggio macchina. Nella Sezione Esercizi si esploreranno molte applicazioni delle pile.

12.6 Le code

Un'altra struttura di dati comune è la *coda* (queue). Una coda è simile a una fila alla cassa di una drogheria: la prima persona della fila sarà servita per prima, mentre gli altri clienti vi entreranno solo dalla sua estremità finale e attenderanno di essere serviti. I nodi delle code possono essere rimossi soltanto dalle loro *teste* e possono essere inseriti soltanto alla *fine* alle stesse. Per questo motivo, una coda è detta anche struttura di dati *FIFO* (*first-in, first-out*; il primo a entrare è il primo a uscire). Le operazioni di inserimento e di rimozione dei nodi sono anche note come *mettere in coda* e *togliere dalla coda*.

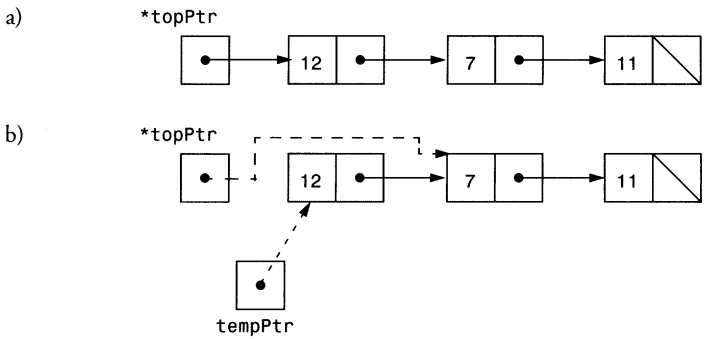


Figura 12.11 L'operazione di pop

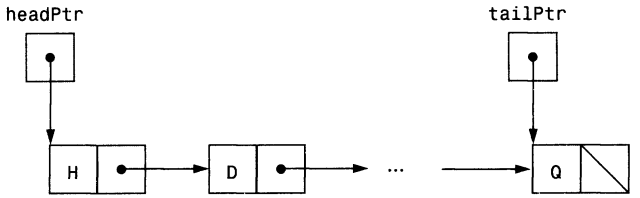


Figura 12.12 Una rappresentazione grafica di una coda

Le code hanno molte applicazioni in informatica. La maggior parte dei computer ha un unico processore e, di conseguenza potrà essere servito un solo utente per volta. Le richieste degli altri utenti saranno quindi immesse in una coda. Ogni richiesta avanzerà gradualmente verso la testa della coda, man mano che gli utenti saranno stati serviti. La richiesta in testa alla coda sarà la prossima a essere servita.

Le code sono utilizzate anche per supportare lo spooling di stampa (la lista dei documenti da stampare). Un ambiente multiutente potrebbe anche avere una sola stampante. Molti utenti potrebbero star generando degli output da stampare. Potrebbero essere generati degli altri output, persino mentre la stampante è occupata. Questi documenti dovranno quindi essere "depositati" su disco, dove attenderanno in una coda che la stampante ritorni disponibile.

Attendono in coda anche i pacchetti di dati che viaggiano nelle reti di computer. Ogni volta che arriva a un nodo della rete, un pacchetto dovrà essere instradato verso quello successivo, lungo un percorso che porta alla destinazione finale del pacchetto. Il nodo di instradamento (il router) è in grado di smistare un pacchetto per volta, perciò gli altri blocchi di dati dovranno essere messi in coda, fino al momento in cui il router potrà instradarli. La Figura 12.12 mostra una coda con diversi nodi. Osservate i puntatori alla testa e alla fine della coda.

Errore tipico 12.7

Non impostare a `NULL` *il membro link nell'ultimo nodo di una coda può provocare degli errori a tempo di esecuzione.*

Il programma della Figura 12.13 (l'output è nella Figura 12.14) eseguirà alcune manipolazioni di una coda. Il programma offrirà diverse scelte: inserire un nodo nella coda (funzione `enqueue`), rimuoverlo dalla stessa (`dequeue`) e terminare l'esecuzione del programma.

```
1  /* Fig. 12.13: fig12_13.c
2     Manipolazione e manutenzione di una coda */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  /* struttura ricorsiva */
8  struct queueNode {
9     char data; /* definisce data come un elemento di tipo char */
10    struct queueNode *nextPtr; /* puntatore a elementi di tipo
                                    queueNode */
```

Figura 12.13 Manipolare una coda (continua)

```
11  }; /* fine della struttura queueNode */
12
13  typedef struct queueNode QueueNode;
14  typedef QueueNode *QueueNodePtr;
15
16  /* prototipi delle funzioni */
17  void printQueue( QueueNodePtr currentPtr );
18  int isEmpty( QueueNodePtr headPtr );
19  char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
20  void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
21                            char value );
22  void instructions( void );
23
24  /* l'esecuzione del programma inizia dalla funzione main */
25  int main()
26  {
27     QueueNodePtr headPtr = NULL; /* inizializza headPtr */
28     QueueNodePtr tailPtr = NULL; /* inizializza tailPtr */
29     int choice; /* scelta del menu da parte dell'utente */
30     char item; /* carattere inserito dall'utente */
31
32     instructions();
33     printf( "? " );
34     scanf( "%d", &choice );
35
36     /* esegue il ciclo finché l'utente non sceglie 3 */
37     while ( choice != 3 ) {
38
39        switch( choice ) {
40
41           /* mette in coda il valore */
42           case 1:
43              printf( "Enter a character: " );
44              scanf( "\n%c", &item );
45              enqueue( &headPtr, &tailPtr, item );
46              printQueue( headPtr );
47              break;
48
49           /* toglie dalla coda il valore */
50           case 2:
51
52              /* se la coda non è vuota */
53              if ( !isEmpty( headPtr ) ) {
54                 item = dequeue( &headPtr, &tailPtr );
55                 printf( "%c has been dequeued.\n", item );
56              } /* fine del comando if */
57
58              printQueue( headPtr );
59              break;
```

Figura 12.13 Manipolare una coda (continua)

```
60
61          default:
62             printf( "Invalid choice.\n\n" );
63             instructions();
64             break;
65
66       } /* fine del comando switch */
67
68       printf( "? " );
69       scanf( "%d", &choice );
70    } /* fine del comando while */
71
72    printf( "End of run.\n" );
73
74    return 0; /* indica che il programma è terminato con successo */
75
76 } /* fine della funzione main */
77
78 /* Visualizza le istruzioni del programma all'utente */
79 void instructions( void )
80 {
81    printf ( "Enter your choice:\n"
82             "   1 to add an item to the queue\n"
83             "   2 to remove an item from the queue\n"
84             "   3 to end\n" );
85 } /* fine della funzione instructions */
86
87 /* inserisce un nodo alla fine della coda */
88 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
89                           char value)
90 {
91    QueueNodePtr newPtr;
92
93    newPtr = malloc( sizeof( QueueNode ) );
94
95    if ( newPtr != NULL ) {
96       newPtr->data = value;
97       newPtr->nextPtr = NULL;
98
99       /* se la coda è vuota, inserisce il nodo in testa */
100       if ( isEmpty( *headPtr ) ) {
101          *headPtr = newPtr;
102       } /* fine del ramo if */
103       else {
104          ( *tailPtr )->nextPtr = newPtr;
105       } /* fine del ramo else */
106
107       *tailPtr = newPtr;
108    } /* fine del ramo if */
```

Figura 12.13 Manipolare una coda (continua)

```
109      else {
110         printf( "%c not inserted. No memory available.\n", value );
111      } /* fine del ramo else */
112
113   } /* fine della funzione enqueue */
114
115   /* rimuove un nodo dalla testa della coda */
116   char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
117   {
118      char value; /* valore del nodo */
119      QueueNodePtr tempPtr; /* puntatore a un nodo temporaneo */
120
121      value = ( *headPtr )->data;
122      tempPtr = *headPtr;
123      *headPtr = ( *headPtr )->nextPtr;
124
125      /* se la coda non è vuota */
126      if ( *headPtr == NULL ) {
127         *tailPtr = NULL;
128      } /* fine del comando if */
129
130      free( tempPtr );
131
132      return value;
133
134   } /* fine della funzione dequeue */
135
136   /* Se la coda è vuota, restituisce 1 altrimenti 0 */
137   int isEmpty( QueueNodePtr headPtr )
138   {
139      return headPtr == NULL;
140
141   } /* fine della funzione isEmpty */
142
143   /* Visualizza la coda */
144   void printQueue( QueueNodePtr currentPtr )
145   {
146
147      /* se la coda non è vuota */
148      if ( currentPtr == NULL ) {
149         printf( "Queue is empty.\n\n" );
150      } /* fine del ramo if */
151      else {
152         printf( "The queue is:\n" );
153
154         /* finché non raggiunge la fine della coda */
155         while ( currentPtr != NULL ) {
156            printf( "%c --> ", currentPtr->data );
157            currentPtr = currentPtr->nextPtr;
```

Figura 12.13 Manipolare una coda (continua)

```
158         } /* fine del comando while */
159
160         printf( "NULL\n\n" );
161      } /* fine del ramo else */
162
163   } /* fine della funzione printQueue */
```

Figura 12.13 Manipolare una coda

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.
```

Figura 12.14 L'output di esempio per il programma della Figura 12.13 (continua)

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Figura 12.14 L'output di esempio per il programma della Figura 12.13

La funzione enqueue (righe 88-113) riceverà da main tre argomenti: gli indirizzi dei puntatori alla testa e alla fine della coda, e il valore da inserire nella stessa. La funzione consisterà di tre passi:

1) Per creare un nuovo nodo: richiamare malloc, assegnare l'indirizzo della memoria allocata a newPtr (riga 93), il valore da inserire nella fila a newPtr->data (riga 96) e NULL a newPtr->nextPtr (riga 97).
2) Nel caso in cui la fila sia vuota (riga 100), assegnare newPtr a *headPtr (riga 101); altrimenti, assegnare il puntatore newPtr a (*tailPtr)->nextPtr (riga 104).
3) Assegnare newPtr a *tailPtr (riga 107).

La Figura 12.15 mostra l'attività di enqueue. La parte a) della figura mostra la coda e il nuovo nodo, prima dell'operazione. Le linee tratteggiate nella parte b) mostrano i passi 2 e 3 della funzione enqueue, che consentiranno al nuovo nodo di essere aggiunto alla fine di una coda non vuota.

La funzione dequeue (righe 116-134) riceverà, come argomenti, gli indirizzi dei puntatori alla testa e alla fine della coda e rimuoverà il primo nodo della stessa. L'attività della dequeue consisterà di sei passi:

1) Assegnare (*headPtr)->data a value per salvare il valore (riga 121).
2) Assegnare *headPtr a tempPtr (riga 122), che sarà utilizzato per rilasciare con free la memoria non più necessaria.

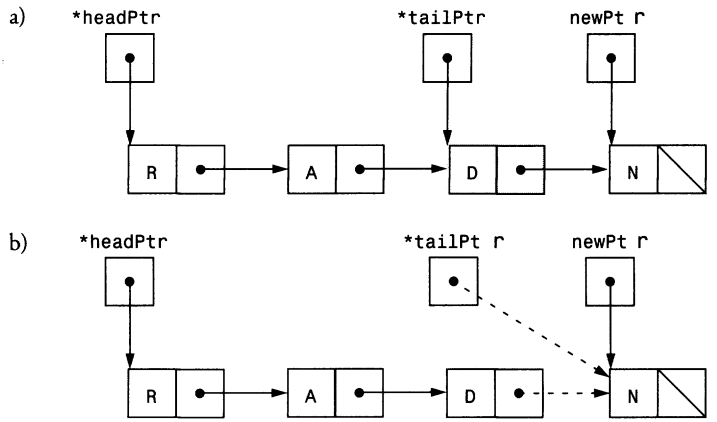


Figura 12.15 Una rappresentazione grafica dell'attività di enqueue

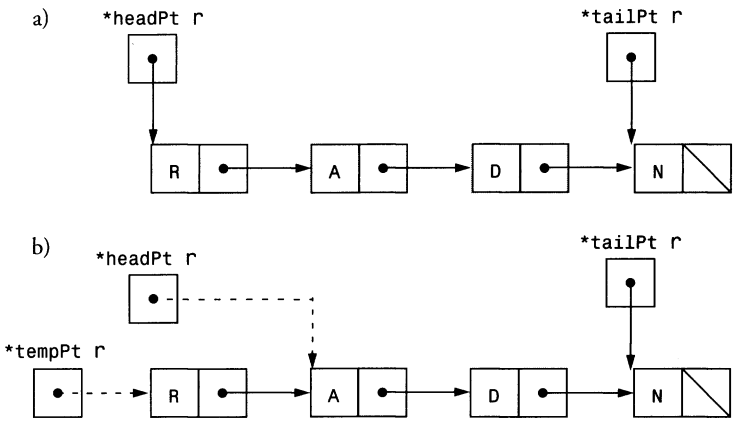


Figura 12.16 Una rappresentazione grafica dell'attività di dequeue.

3) Assegnare `(*headPtr)->nextPtr` a `*headPtr` (riga 123) in modo che `*headPtr` punti così al nuovo primo nodo della coda.
4) Nel caso in cui `*headPtr` sia `NULL` (riga 126), assegnare `NULL` a `*tailPtr` (riga 127).
5) Rilasciare la memoria puntata da `tempPtr` (riga 130).
6) Restituire `value` al chiamante (riga 132).

La Figura 12.16 mostra la funzione `dequeue`. La parte a) mostra la coda, dopo il precedente intervento di `enqueue`. La parte b) mostra `tempPtr` mentre punta al nodo rimosso dalla coda, mentre `headPtr` punta al suo nuovo primo nodo. La funzione `free` sarà usata per ricuperare la memoria puntata da `tempPtr`.

12.7 Gli alberi

Le liste concatenate, le pile e le code sono *strutture di dati lineari*. Un albero è invece una struttura di dati non lineare, bidimensionale con delle proprietà speciali. I nodi di un albero contengono due o più membri link. Questa sezione discuterà degli *alberi binari* (Figura 12.17): ovverosia, degli alberi i cui nodi contengano solo due link (che potrebbero essere entrambi `NULL`). Il *nodo radice* è il primo di ogni albero. Ogni link del nodo radice punta a un *figlio*. Il *figlio sinistro* è il primo nodo del *sottoalbero sinistro*, mentre il *figlio destro* è il primo nodo del *sottoalbero destro*. I figli di uno stesso nodo sono detti *fratelli*. Un nodo senza figli è detto *nodo foglia*. Normalmente, gli informatici disegnano gli alberi cominciando dal nodo radice e procedendo verso il basso: esattamente in modo inverso agli alberi che si trovano in natura.

In questa sezione, sarà creato un albero binario speciale chiamato *albero di ricerca binaria*. La caratteristica di un albero di ricerca binaria, oltre all'assenza di nodi duplicati, è che i valori contenuti nei nodi di ogni sottoalbero sinistro sono inferiori al valore contenuto nel rispettivo nodo padre, mentre quelli di ogni sottoalbero destro sono maggiori. La Figura 12.18 mostra un albero di ricerca binaria con 12 valori. Osservate che la forma di un albero di ricerca binaria che corrisponde a un insieme di dati può variare, secondo l'ordine in cui i valori sono inseriti nell'albero.

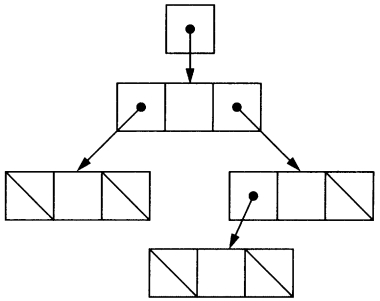


Figura 12.17 Una rappresentazione grafica di un albero binario

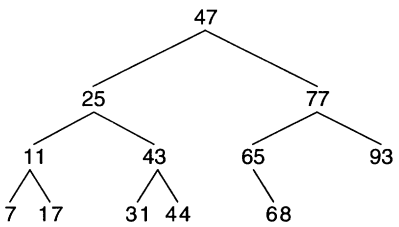


Figura 12.18 Un albero di ricerca binaria

Errore tipico 12.8

Non impostare a `NULL` *i link dei nodi foglia di un albero può provocare degli errori a tempo di esecuzione.*

Il programma della Figura 12.19 (l'output è mostrato in Figura 12.20) creerà un albero di ricerca binaria e lo visiterà in tre modi: *in ordine*, ovverosia, con una *visita simmetrica*, *anticipata* e *differita*. Il programma genererà 10 numeri casuali e li inserirà nell'albero, escludendo i valori duplicati.

Le funzioni usate nella Figura 12.19 per creare e visitare un albero di ricerca binaria sono ricorsive. La funzione `insertNode` (righe 58-93) riceverà come argomenti l'indirizzo dell'albero e l'intero che vi dovrà essere immagazzinato. *Un nodo può essere inserito in un albero di ricerca binaria soltanto come nodo foglia.* I passi per inserire un nodo in un albero di ricerca binaria saranno i seguenti:

1) Nel caso in cui `*treePtr` sia `NULL` (riga 62), creare un nuovo nodo (riga 63). Richiamare `malloc`, assegnare la memoria allocata a `*treePtr`, l'intero da immagazzinare a `(*treePtr)->data` (riga 67), `NULL` a `(*treePtr)->leftPtr` e `(*treePtr)->rightPtr` (righe 68-69) e restituire il controllo al chiamante (la funzione `main` o una precedente chiamata a `insertNode`).
2) Nel caso in cui il valore di `*treePtr` non sia `NULL` e quello da inserire sia inferiore a `(*treePtr)->data`, la funzione `insertNode` sarà chiamata con l'indirizzo di `(*treePtr)->leftPtr` (riga 80). Nel caso in cui il valore da inserire sia maggiore di `(*treePtr)->data`, la funzione `insertNode` sarà chiamata con l'indirizzo di `(*treePtr)->rightPtr` (riga 85). Altrimenti i passi ricorsivi continueranno, fino al momento in cui sarà stato ritrovato un puntatore `NULL` e, quindi, sarà eseguito il passo 1) per inserire il nuovo nodo.

```
1  /* Fig. 12.19: fig12_19.c
2     Creare un albero binario ed effettuare
3     una visita anticipata, simmetrica e differita dello stesso */
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  /* struttura ricorsiva */
9  struct treeNode {
10    struct treeNode *leftPtr; /* puntatore al sottoalbero sinistro */
11    int data; /* valore del nodo */
12    struct treeNode *rightPtr; /* puntatore al sottoalbero destro */
13 }; /* fine della struttura treeNode */
14
15 typedef struct treeNode TreeNode; /* sinonimo di struct treeNode */
16 typedef TreeNode *TreeNodePtr; /* sinonimo di TreeNode* */
17
18 /* prototipi */
19 void insertNode( TreeNodePtr *treePtr, int value );
20 void inOrder( TreeNodePtr treePtr );
21 void preOrder( TreeNodePtr treePtr );
22 void postOrder( TreeNodePtr treePtr );
23
24 /* l'esecuzione del programma inizia dalla funzione main */
25 int main()
26 {
27    int i; /* contatore per eseguire cicli da 1 a 10 */
28    int item; /* variabile per memorizzare valori casuali */
29    TreeNodePtr rootPtr = NULL; /* albero inizialmente vuoto */
30
31    srand( time( NULL ) );
32    printf( "The numbers being placed in the tree are:\n" );
33
34    /* inserisce nell'albero valori casuali compresi tra 1 e 15 */
35    for ( i = 1; i <= 10; i++ ) {
36       item = rand() % 15;
37       printf( "%3d", item );
38       insertNode( &rootPtr, item );
39    } /* fine del comando for */
40
41    /* visita anticipata dell'albero */
42    printf( "\n\nThe preOrder traversal is:\n" );
43    preOrder( rootPtr );
44
45    /* visita simmetrica dell'albero */
46    printf( "\n\nThe inOrder traversal is:\n" );
47    inOrder( rootPtr );
48
```

Figura 12.19 Creare e visitare un albero binario (continua)

```
49       /* visita differita dell'albero */
50       printf( "\n\nThe postOrder traversal is:\n" );
51       postOrder( rootPtr );
52
53       return 0; /* indica che il programma è terminato con successo */
54
55    } /* fine della funzione main */
56
57    /* inserisce un nodo nell'albero */
58    void insertNode( TreeNodePtr *treePtr, int value )
59    {
60
61       /* se l'albero è vuoto */
62       if ( *treePtr == NULL ) {
63          *treePtr = malloc( sizeof( TREENODE ) );
64
65          /* se la memoria è stata allocata, memorizza i dati */
66          if ( *treePtr != NULL ) {
67             ( *treePtr )->data = value;
68             ( *treePtr )->leftPtr = NULL;
69             ( *treePtr )->rightPtr = NULL;
70          } /* fine del ramo if */
71          else {
72             printf( "%d not inserted. No memory available.\n", value );
73          } /* fine del ramo else */
74
75       } /* fine del ramo if */
76       else { /* l'albero non è vuoto */
77
78          /* il dato da inserire è minore del dato del nodo corrente */
79          if ( value < ( *treePtr )->data ) {
80             insertNode( &( ( *treePtr )->leftPtr ), value );
81          } /* fine del ramo if */
82
83          /* il dato da inserire è maggiore del dato del nodo corrente */
84          else if ( value > ( *treePtr )->data ) {
85             insertNode( &( ( *treePtr )->rightPtr ), value );
86          } /* fine del ramo else if */
87          else { /* se il dato è un duplicato, viene ignorato */
88             printf( "dup" );
89          } /* fine del ramo else */
90
91       } /* fine del ramo else */
92
93    } /* fine della funzione insertNode */
94
95    /* inizia la visita simmetrica dell'albero */
96    void inOrder( TreeNodePtr treePtr )
```

Figura 12.19 Creare e visitare un albero binario (continua)

```
97   {
98
99      /* se l'albero non è vuoto, viene "attraversato" */
100     if ( treePtr != NULL ) {
101        inOrder( treePtr->leftPtr );
102        printf( "%3d", treePtr->data );
103        inOrder( treePtr->rightPtr );
104     } /* fine del comando if */
105
106  } /* fine della funzione inOrder */
107
108  /* inizia la visita anticipata dell'albero */
109  void preOrder( TreeNodePtr treePtr )
110  {
111
112     /* se l'albero non è vuoto, viene "attraversato" */
113     if ( treePtr != NULL ) {
114        printf( "%3d", treePtr->data );
115        preOrder( treePtr->leftPtr );
116        preOrder( treePtr->rightPtr );
117     } /* fine del comando if */
118
119  } /* fine della funzione preOrder */
120
121  /* inizia la visita differita dell'albero */
122  void postOrder( TreeNodePtr treePtr )
123  {
124
125     /* se l'albero non è vuoto, viene "attraversato" */
126     if ( treePtr != NULL ) {
127        postOrder( treePtr->leftPtr );
128        postOrder( treePtr->rightPtr );
129        printf( "%3d", treePtr->data );
130     } /* fine del comando if */
131
132  } /* fine della funzione postOrder */
```

Figura 12.19 Creare e visitare un albero binario

```
The numbers being placed in the tree are:
  6   7   4   12 7dup   2   2dup 5   7dup   11

The preOrder traversal is:
  6   4   2   5   7 12 11

The inOrder traversal is:
  2   4   5   6   7 11 12

The postOrder traversal is:
  2   5   4 11 12   7   6
```

Figura 12.20 L'output di esempio per il programma in Figura 12.19

Le funzioni `inOrder` (righe 96-106), `preOrder` (righe 109-119) e `postOrder` (righe 122-132) riceveranno un albero (ovverosia, il puntatore al suo nodo radice) e lo visiteranno.

I passi per una visita simmetrica con `inOrder` saranno:

1) Visitare il sottoalbero sinistro con `inOrder`.
2) Elaborare il valore nel nodo.
3) Visitare il sottoalbero destro con `inOrder`.

Il valore di un nodo non sarà elaborato, fin quando non saranno stati elaborati quelli del suo sottoalbero sinistro. La visita simmetrica, con `inOrder`, dell'albero in Figura 12.21 sarà:

```
6 13 17 27 33 42 48
```

Osservate che la visita simmetrica, con `inOrder`, di un albero di ricerca binaria visualizzerà i valori dei nodi in ordine crescente. Il processo di creazione di un albero di ricerca binaria, in realtà, ordina i dati e, per questo motivo, è chiamato *ordinamento dell'albero binario.*

I passi per una visita anticipata, con `preOrder`, saranno:

1) Elaborare il valore del nodo.
2) Visitare il sottoalbero sinistro con `preOrder`.
3) Visitare il sottoalbero destro con `preOrder`.

Il valore di ogni nodo sarà elaborato nel momento stesso in cui sarà visitato. Una volta che il valore di un dato nodo sarà stato elaborato, saranno elaborati prima i valori del sottoalbero sinistro e quindi quelli del sottoalbero destro. La visita anticipata, con `preOrder`, dell'albero in Figura 12.21 sarà:

```
27 13 6 17 42 33 48
```

I passi per una visita differita, con `postOrder`, saranno:

1) Visitare il sottoalbero sinistro con `postOrder`.
2) Vistare il sottoalbero destro con `postOrder`.
3) Elaborare il valore del nodo.

Il valore di ogni nodo non sarà visualizzato, fin quando non saranno stati visualizzati i valori dei propri figli. La visita differita, con `postOrder`, dell'albero in Figura 12.21 sarà:

```
6 17 13 33 48 42 27
```

L'albero di ricerca binaria facilita *l'eliminazione dei duplicati.* Durante la creazione dell'albero, sarà riconosciuto ogni tentativo di inserimento di un valore duplicato, perché questo seguirà, a ogni confronto, le stesse decisioni "vai a sinistra" o "vai a destra" del valore originario. Di conseguenza, alla fine il valore duplicato sarà confrontato con un nodo contenente lo stesso dato. A quel punto, il valore duplicato potrà semplicemente essere eliminato.

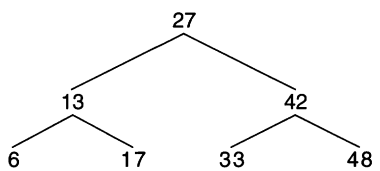


Figura 12.21 Un albero di ricerca binaria

Anche la ricerca di un valore in un albero binario è un'operazione rapida. Qualora l'albero sia stato riempito per bene, allora ogni livello conterrà circa il doppio degli elementi di quello precedente. Di conseguenza, un albero di ricerca binaria con n elementi avrà un massimo di livelli e, quindi, dovranno essere eseguiti un massimo di confronti per trovare una corrispondenza o per determinare che non ne esistano. Ciò significa, per esempio, che per un'indagine in un albero di ricerca binaria (riempito per bene) con 1.000 elementi, dovranno essere eseguiti non più di 10 confronti, perché $2^{10} > 1.000$. Per un'indagine in un albero di ricerca binaria (riempito per bene) con 1.000.000 elementi, dovranno essere eseguiti non più di 20 confronti, perché $2^{20} > 1.000.000$.

Nella sezione Esercizi, saranno presentati degli algoritmi per molte altre operazioni sugli alberi binari, come l'eliminazione di un elemento, la visualizzazione in un formato ad albero bidimensionale e l'esecuzione di una visita per livelli. La visita per livelli di un albero binario ispezionerà i suoi nodi una riga per volta, incominciando dal livello del nodo radice. Per ogni livello dell'albero, i nodi saranno visitati da sinistra a destra. Altri esercizi sugli alberi binari comprenderanno: l'accettazione di valori duplicati, l'inserimento di quelli di tipo stringa e la determinazione del numero dei livelli contenuti nell'albero binario.

Esercizi di autovalutazione

12.1 Riempite gli spazi vuoti in ognuna delle seguenti righe:

a) Una struttura _____ è usata per formare strutture di dati dinamiche.
b) La funzione _____ è utilizzata per allocare dinamicamente la memoria.
c) Una _____ è una versione specializzata di una lista concatenata, in cui i nodi possono essere inseriti ed eliminati soltanto dall'inizio della lista.
d) Le funzioni che danno uno "sguardo" a una lista concatenata, ma che non la modificano sono dette _____.
e) Una coda è una struttura di dati _____.
f) Il puntatore al nodo successivo in una lista concatenata è detto anche _____.
g) La funzione _____ è usata per recuperare la memoria allocata dinamicamente.
h) Una _____ è una versione specializzata di una lista concatenata, in cui i nodi possono essere inseriti soltanto alla fine della lista e rimossi soltanto dall'inizio della stessa.
i) Un _____ è una struttura di dati non lineare, bidimensionale che contiene nodi con due o più collegamenti.
j) Una pila è una struttura di dati _____, perché l'ultimo nodo inserito sarà il primo a essere rimosso.
k) I nodi di un albero _____ contengono due membri link.
l) Il primo nodo di un albero è il nodo _____.
m) Nel nodo di un albero, ogni link punta a un _____ o _____ di quel nodo.
n) Un nodo di un albero che non ha figli è detto nodo _____.
o) I tre algoritmi di visita (discussi in questo capitolo) per un albero binario sono la visita _____, _____ e _____.

12.2 Quali sono le differenze tra una lista concatenata e una pila?

12.3 Quali sono le differenze tra una pila e una coda?

12.4 Scrivete un'istruzione o un insieme di istruzioni che eseguano ognuno dei seguenti compiti. Supponete che tutte le manipolazioni debbano essere eseguite nella funzione `main`, di conseguenza non sarà necessario alcun indirizzo di variabile puntatore, e che siano state effettuate le seguenti definizioni:

```
struct gradeNode {
   char lastName[ 20 ];
   double grade;
   struct gradeNode *nextPtr;
};

typedef struct gradeNode GradeNode;
typedef GRADENODE *GradeNodePtr;
```

a) Create un puntatore all'inizio della lista chiamato `startPtr`. La lista è vuota.
b) Create un nuovo nodo di tipo `GradeNode`, che sia oggetto di riferimento da parte del puntatore `newPtr` di tipo `GradeNodePtr`. Assegnate la stringa `"Jones"` al membro `lastName` e il valore `91,5` al membro `grade` (usate `strcpy`). Fornite ogni dichiarazione e istruzione necessaria.
c) Supponete che la lista puntata da `startPtr` consista al momento di due nodi, uno contenente `"Jones"` e l'altro `"Smith"`, e che questi siano in ordine alfabetico. Fornite le istruzioni necessarie per inserire ordinatamente dei nodi che contengano i seguenti dati per `lastName` e `grade`:

```
"Adams"            85,0
"Thompson          73,5
"Pritchard"  66,5
```

   Per eseguire gli inserimenti, usate i puntatori `previousPtr`, `currentPtr` e `newPtr`. Stabilite a cosa punteranno `previousPtr` e `currentPtr` prima di ogni inserimento. Supponete che `newPtr` punti sempre al nuovo nodo e che a questo siano già stati assegnati i dati.
d) Scrivete un ciclo `while` che visualizzi i dati contenuti in ogni nodo della lista. Usate il puntatore `currentPtr` per scorrere la lista.
e) Scrivete un ciclo `while` che elimini tutti i nodi della lista e rilasci la memoria associata a ognuno di essi. Usate i puntatori `currentPtr` e `tempPtr` per muovervi lungo la lista e per rilasciare la memoria.

12.5 Riportate manualmente la visita simmetrica, anticipata e differita dell'albero di ricerca binaria mostrato nella Figura 12.22.

Risposte agli esercizi di autovalutazione

12.1 a) ricorsiva. b) `malloc`. c) pila. d) predicati. e) FIFO. f) link. g) `free`. h) coda. i) albero. j) LIFO. k) binario. l) radice. m) figlio, sottoalbero. n) foglia. o) simmetrica, anticipata, differita.

12.2 In una lista concatenata, è possibile inserire ed eliminare un nodo in qualsiasi posizione. I nodi di una pila, invece, possono essere inseriti e rimossi soltanto dalla sua testa.

12.3 Una coda ha puntatori alla testa e alla fine della stessa, così che i nodi possano essere inseriti alla fine e rimossi dalla testa. Una pila ha solo un puntatore alla sua testa, dove sono eseguite le operazioni di inserimento e di rimozione.

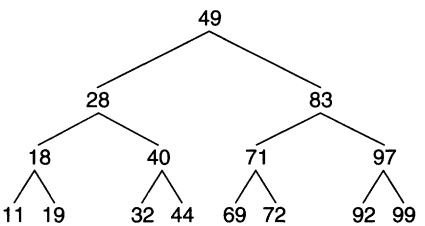


Figura 12.22 Un albero di ricerca binaria con 15 nodi

12.4 a)
```
GradeNodePtr startPtr = NULL;
```
b)
```
GradeNodePtr newPtr;
newPtr = malloc( sizeof( GradeNode ) );
strcpy( newPtr->lastName, "Jones" );
newPtr->grade = 91.5;
newPtr->nextPtr = NULL;
```
c) Per inserire `"Adams"`:
`previousPtr` è `NULL`, `currentPtr` punta al primo elemento della lista.
```
newPtr->nextPtr = currentPtr;
startPtr = newPtr;
```
Per inserire `"Thompson"`:
`previousPtr` punta all'ultimo elemento della lista (contenente `"Smith"`)
`currentPtr` è `NULL`.
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
Per inserire `"Pritchard"`:
`previousPtr` punta al nodo contenente `"Jones"`
`currentPtr` punta al nodo contenente `"Smith"`.
```
newPtr->nextPtr = currentPtr;
previousPtr->nextPtr = newPtr;
```
d)
```
currentPtr = startPtr;
while ( currentPtr != NULL ) {
   printf( "Lastname = %s\nGrade = %6.2f\n",
          currentPtr->lastName, currentPtr->grade );
   currentPtr = currentPtr->nextPtr;
}
```
e)
```
currentPtr = startPtr;
while ( currentPtr != NULL ) {
   tempPtr = currentPtr;
   currentPtr = currentPtr->nextPtr;
   free( tempPtr );
}
startPtr = NULL;
```

12.5 La visita simmetrica è:

```
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99
```

La visita anticipata è:

```
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99
```

La visita differita è:

```
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49
```

Esercizi

12.6 Scrivete un programma che colleghi due liste concatenate di caratteri. Il programma dovrà includere la funzione `concatenate`, che prenda come argomenti i puntatori a entrambe le liste e concateni la seconda alla prima.

12.7 Scrivete un programma che unisca due liste ordinate di valori interi in una singola lista ordinata di valori interi. La funzione `merge` dovrà ricevere i puntatori al primo nodo di ognuna delle liste da unire, e dovrà restituire un puntatore al primo nodo della lista unita.

12.8 Scrivete un programma che inserisca ordinatamente in una lista concatenata 25 interi casuali compresi tra zero e 100. Il programma dovrà calcolare la somma degli elementi e la loro media con un valore in virgola mobile.

12.9 Scrivete un programma che crei una lista concatenata di 10 caratteri e, in seguito, create una copia della lista con gli elementi in ordine inverso.

12.10 Scrivete un programma che prenda in input una linea di testo, e utilizzi una pila per visualizzare la linea in ordine inverso.

12.11 Scrivete un programma che usi una pila per determinare se una stringa è palindroma (ovverosia, se possa essere letta allo stesso modo in entrambi i sensi). Il programma dovrà ignorare gli spazi e la punteggiatura.

12.12 Le pile sono usate dai compilatori come supporto per il processo di valutazione delle espressioni e di generazione del codice in linguaggio macchina. In questo e nel prossimo esercizio, esamineremo con cura il modo in cui i compilatori valutano delle espressioni aritmetiche che consistano solamente di costanti, operatori e parentesi.

Gli esseri umani scrivono generalmente delle espressioni come `3 + 4` e `7 / 9`, in cui l'operatore (`+` o `/`, in questo caso) è scritto tra i suoi operandi, ovverosia in *notazione infissa*. I computer "preferiscono" invece la *notazione polacca inversa*, con l'operatore scritto a destra dei suoi due operandi. Con la notazione polacca inversa, le due precedenti espressioni infisse apparirebbero rispettivamente come `3 4 +` e `7 9 /`.

Per valutare un'espressione in notazione infissa complessa, il compilatore la convertirà prima nella notazione polacca inversa e quindi la valuterà in questa sua nuova versione. Ognuno di questi algoritmi richiede soltanto un singolo passaggio da sinistra a destra sull'espressione. Ogni algoritmo utilizza una pila per supportare la sua attività, anche se in ognuno di essi la pila è usata per uno scopo diverso.

In questo esercizio scriverete una versione dell'algoritmo di conversione da notazione infissa a polacca inversa. Nel prossimo esercizio, scriverete una versione dell'algoritmo di valutazione di un'espressione in notazione polacca inversa.

Scrivete un programma che converta un'ordinaria espressione aritmetica in notazione infissa (supponete che ne sia stata immessa una valida), formata da interi di una sola cifra come

```
(6 + 2) * 5 - 8 / 4
```

in un'espressione in notazione polacca inversa. La versione polacca inversa della precedente espressione in notazione infissa è

```
6 2 + 5 * 8 4 / -
```

Il programma dovrà leggere l'espressione e immagazzinarla nel vettore di caratteri `infix` e, in seguito, utilizzare la versione modificata delle funzioni per la gestione di una pila implementate in questo capitolo, per aiutare a creare l'espressione in notazione polacca inversa nel vettore di caratteri `postfix`. L'algoritmo per creare un'espressione in notazione polacca inversa è il seguente:

1) Inserite nella pila una parentesi aperta `'('`.
2) Accodate alla fine di `infix` una parentesi chiusa `')'`.
3) Fintanto che la pila non è vuota, leggete `infix` da sinistra a destra ed eseguite le seguenti operazioni:

   Nel caso in cui il carattere corrente di `infix` sia un numero, copiatelo nell'elemento successivo di `postfix`.

   Nel caso in cui il carattere corrente di `infix` sia una parentesi aperta, inseritela nella pila.

   Nel caso in cui il carattere corrente di `infix` sia un operatore,

   Estraete gli operatori (se ce ne sono) dalla testa della pila, fintanto che abbiano una priorità maggiore o uguale a quella dell'operatore corrente, e inseriteli in `postfix`.

   Inserite nella pila il carattere corrente in `infix`.

Nel caso in cui il carattere corrente di `infix` sia una parentesi chiusa,

Estraete gli operatori dalla testa della pila e inseriteli in `postfix`, finché non ci sarà una parentesi aperta sulla cima della pila.

Estraete (ed eliminate) la parentesi aperta dalla pila.

In un'espressione dovranno essere consentite le seguenti operazioni aritmetiche:

- `+` addizione
- `-` sottrazione
- `*` moltiplicazione
- `/` divisione
- `^` elevamento a potenza
- `%` modulo

La pila dovrà essere gestita con le seguenti dichiarazioni:

```
struct stackNode {
   char data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef STACKNODE *StackNodePtr;
```

Il programma dovrà consistere di una funzione **main** e di altre otto funzioni, con le seguenti intestazioni:

```
void convertToPostfix( char infix[], char postfix[] )
```

Convertirà la notazione infissa in quella polacca inversa.

```
int isOperator( char c )
```

Determinerà se `c` è un operatore.

```
int precedence( char operator1, char operator2 )
```

Determinerà se la priorità di `operator1` è minore, uguale o maggiore di quella di `operator2`. La funzione restituirà rispettivamente –1, 0 o 1.

```
void push( StackNodePtr *topPtr, char value )
```

Inserirà un valore in cima alla pila.

```
char pop( StackNodePtr *topPtr )
```

Estrarrà un valore dalla cima della pila.

```
char stackTop( StackNodePtr topPtr )
```

Restituirà il valore contenuto in cima alla pila, senza estrarlo dalla stessa.

```
int isEmpty( StackNodePtr topPtr )
```

Determinerà se la pila è vuota.

```
void printStack( StackNodePtr topPtr )
```

Visualizzerà la pila.

12.13 Scrivete un programma che valuti un'espressione in notazione polacca inversa (supponete che sia valida) come

```
6 2 + 5 * 8 4 / -
```

Il programma dovrà leggere e immagazzinare in un vettore un'espressione in notazione polacca inversa formata da numeri e operatori. Usando la versione modificata delle funzioni per la gestione di una pila implementate in precedenza in questo capitolo, il programma dovrà analizzare l'espressione e valutarla. L'algoritmo è il seguente:

1) Accodate il carattere nullo (`'\0'`) alla fine dell'espressione in notazione polacca inversa. Nel momento in cui il carattere nullo sarà stato incontrato, non saranno necessarie ulteriori operazioni.
2) Fintanto che `'\0'` non sia stato incontrato, leggete l'espressione da sinistra a destra.

   Nel caso in cui il carattere corrente sia un numero, inserite il suo valore intero in testa alla pila. Ricordate che il valore intero di un carattere numerico è quello assunto nell'insieme dei caratteri del computer meno il valore di `'0'`.

   Altrimenti, qualora il carattere corrente sia un *operatore*, estraete dalla cima della pila i primi due elementi e immagazzinateli nelle variabili `x` e `y`.

      Calcolate `y` *operatore* `x`.

      Inserite il risultato del calcolo in testa alla pila.
3) Estraete il valore contenuto in cima alla pila, qualora nell'espressione abbiate incontrato il carattere nullo. Quel valore sarà proprio il risultato dell'espressione in notazione polacca inversa.

[*Nota*: nel precedente punto 2), quando l'operatore sarà `'/'`, la testa della pila conterrà `2` e il suo elemento successivo sarà `8`, allora dovrete estrarre il `2` in `x`, il valore `8` in `y`, calcolare `8 / 2` e reinserire nella pila il risultato, `4`. Questa nota si applicherà anche all'operatore `'-'`.] Ecco un elenco delle operazioni aritmetiche che saranno consentite in un'espressione:

`+`	addizione
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione
`^`	elevamento a potenza
`%`	modulo

La pila dovrà essere gestita con le seguenti dichiarazioni:

```
struct stackNode {
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef STACKNODE *StackNodePtr;
```

Il programma dovrà essere formato dal `main` e da altre sei funzioni con le seguenti intestazioni:

```
int evaluatePostfixExpression( char *expr )
```

Valuterà l'espressione in notazione polacca inversa.

```
int calculate( int op1, int op2, char operator )
```

Calcolerà il valore dell'espressione `op1 operator op2`.

```
void push(StackNodePtr *topPtr, int value)
```

Inserirà un valore in cima alla pila.

```
int pop(StackNodePtr *topPtr)
```

Estrarrà un valore dalla cima della pila.

`int isEmpty(StackNodePtr topPtr)`

Determinerà se la pila è vuota.

`void printStack(StackNodePtr topPtr)`

Visualizzerà la pila.

12.14 Modificate il programma per la valutazione delle espressioni in notazione polacca inversa dell'Esercizio 12.13, in modo che possa elaborare degli operandi interi maggiori di nove.

12.15 (*Simulazione di un supermercato*) Scrivete un programma che simuli una fila alla cassa di un supermercato. La fila è una coda. I clienti arriveranno casualmente in intervalli interi compresi tra 1 e 4 minuti. Anche i clienti saranno serviti in modo casuale a intervalli interi compresi tra 1 e 4 minuti. Ovviamente, i ritmi dovranno essere bilanciati. La coda crescerà all'infinito qualora il ritmo degli arrivi sia superiore a quello del servizio. Nonostante i ritmi bilanciati, il caso potrebbe causare delle lunghe code. Eseguite la simulazione del supermercato per una giornata lavorativa di 12 ore (720 minuti), usando il seguente algoritmo:

1) Scegliete un intero casuale compreso tra 1 e 4, per determinare il minuto in cui arriverà il primo cliente.
2) Nel momento in cui arriva il primo cliente:

   Determinate il tempo di servizio del cliente (un intero casuale compreso tra 1 e 4);

   Cominciate a servire il cliente;

   Stabilite il tempo di arrivo del prossimo cliente (un intero casuale compreso tra 1 e 4 aggiunto al tempo corrente).
3) Per ogni minuto del giorno:

   Qualora arrivi il cliente successivo:

   Mettete il cliente nella coda;

   Stabilite il tempo di arrivo del prossimo cliente;

   Nel caso in cui sia stato completato il servizio per l'ultimo cliente:

   Comunicatelo

   Togliete dalla coda il prossimo cliente da servire

   Determinate il tempo di completamento del servizio per il cliente (ovverosia, un intero casuale da 1 a 4 aggiunto al tempo corrente).

Eseguite ora la vostra simulazione e ponetevi le seguenti questioni:

a) Qual è stato il numero massimo di clienti in coda durante tutta la simulazione?
b) Qual è stata l'attesa più lunga tra quelle sperimentate dai clienti?
c) Che cosa succederebbe se l'intervallo di arrivo, compreso tra 1 e 4 minuti, fosse cambiato in uno compreso tra 1 e 3 minuti?

12.16 Modificate il programma della Figura 12.19 in modo da consentire l'inserimento di valori duplicati nell'albero binario.

12.17 Scrivete un programma basato su quello della Figura 12.19 che prenda in input una linea di testo, separi la frase in parole, le inserisca in un albero di ricerca binaria e visualizzi la visita simmetrica, anticipata e differita dell'albero.

[*Suggerimento*: leggete la linea di testo e immagazzinatela in un vettore. Usate `strtok` per dividere in parole il testo. Nel momento in cui avrete isolato una parola dovrete creare un nuovo nodo nell'albero, assegnare il puntatore restituito da `strtok` al membro `string` del nuovo nodo e inserirlo nell'albero.]

12.18 In questo capitolo, abbiamo visto che l'eliminazione dei duplicati è semplice, qualora si stia creando un albero di ricerca binaria. Descrivete in che modo eseguireste l'eliminazione dei duplicati usando soltanto un vettore a una dimensione. Confrontate le prestazioni dell'operazione di eliminazione dei duplicati basata sul vettore con quella basata sull'albero di ricerca binaria.

12.19 Scrivete una funzione `depth` che riceva un albero binario e determini di quanti livelli sia composto.

12.20 (*Visualizzazione ricorsiva di una lista in ordine inverso*) Scrivete una funzione `printListBackwards` che visualizzi in modo ricorsivo e in ordine inverso gli elementi di una lista. Usate la vostra funzione in un programma di prova che crei una lista ordinata di interi e la visualizzi in ordine inverso.

12.21 (*Ricerca ricorsiva in una lista*) Scrivete una funzione `searchList` che cerchi in modo ricorsivo un dato valore all'interno di una lista concatenata. La funzione dovrà restituire un puntatore al valore, qualora l'abbia ritrovato, o `NULL` in caso contrario. Usate la vostra funzione in un programma di prova che crei una lista di interi. Il programma dovrà richiedere all'utente il valore da individuare nella lista.

12.22 (*Rimozione da un albero binario*) In questo esercizio, affronteremo la rimozione degli elementi dagli alberi di ricerca binaria. L'algoritmo di rimozione non è così semplice come quello di inserimento. Durante l'eliminazione di un elemento, ci si potrebbe imbattere in tre casi: l'elemento è contenuto in un nodo foglia (ovverosia, non ha figli), oppure in un nodo che ha un figlio, o in un nodo che ha due figli.

Qualora l'elemento da eliminare sia contenuto in una foglia, questo sarà rimosso e il puntatore del nodo padre sarà impostato a `NULL`.

Qualora l'elemento da eliminare sia contenuto in un nodo che abbia un figlio, il puntatore del padre sarà impostato in modo da fare riferimento al nodo figlio, mentre quello che contiene i dati sarà eliminato. Ciò significa che, all'interno dell'albero, il nodo figlio assumerà il posto di quello rimosso. L'ultimo caso è il più difficile. Qualora si elimini uno che abbia due figli, un altro nodo dell'albero dovrà assumere il suo posto. Tuttavia, il puntatore del nodo padre non potrà semplicemente essere impostato in modo da fare riferimento a uno dei figli di quello da eliminare. Ciò perché, nella maggior parte dei casi, l'albero di ricerca binaria risultante non si manterrebbe fedele alla sua caratteristica principale: *I valori di ogni sottoalbero sinistro dell'albero sono inferiori a quello del nodo padre, mentre quelli di ogni sottoalbero destro dell'albero sono maggiori di quello del nodo padre.*

Quale nodo dovrà essere usato come *nodo sostitutivo* per mantenere questa caratteristica? Potrà essere il nodo contenente il valore più grande dell'albero che sia inferiore a quello del nodo eliminato; oppure potrà essere quello contenente il valore più piccolo dell'albero che sia maggiore di quello del nodo eliminato. Consideriamo il nodo con il valore inferiore. In un albero di ricerca binaria, il valore più grande inferiore a quello di un padre sarà sistemato nel sottoalbero sinistro del nodo padre, e sarà sicuramente contenuto nel nodo all'estremità destra del sottoalbero. Troveremo il suddetto nodo scendendo nel sottoalbero sinistro del padre e andando sempre a destra, fintanto che il puntatore al figlio destro del nodo corrente non sarà `NULL`. A questo punto, staremo puntando al nodo di sostituzione che potrà essere una foglia, o un nodo con un solo figlio alla sua sinistra. Qualora il nodo di sostituzione sia una foglia, i passi per eseguire l'eliminazione saranno i seguenti:

1) Salvate il puntatore al nodo da eliminare in una variabile temporanea di tipo puntatore: questo puntatore sarà usato per liberare la memoria allocata dinamicamente.
2) Impostate il puntatore del padre del nodo da eliminare in modo che faccia riferimento al nodo di sostituzione.
3) Impostate al valore nullo il puntatore al sottoalbero destro nel padre del nodo di sostituzione.
4) Impostate il puntatore al sottoalbero destro (sinistro) del nodo di sostituzione in modo che faccia riferimento al sottoalbero destro (sinistro) del nodo da eliminare.
5) Eliminate il nodo puntato dalla variabile temporanea di tipo puntatore.

Per eseguire l'eliminazione nel caso di un nodo di sostituzione che abbia un figlio sinistro, si effettueranno delle operazioni simili a quelle effettuate nel caso di un nodo di sostituzione senza figli, ma l'algoritmo dovrà spostare anche il figlio nella posizione attualmente occupata dal nodo di sostituzione.

Qualora quello di sostituzione sia un nodo con un figlio a sinistra, i passi per eseguire l'eliminazione saranno i seguenti:

1) Salvate il puntatore al nodo da eliminare in una variabile temporanea di tipo puntatore.
2) Impostate il puntatore del padre del nodo da eliminare in modo che faccia riferimento al nodo di sostituzione.
3) Impostate il puntatore al sottoalbero destro contenuto nel padre del nodo di sostituzione, in modo che faccia riferimento al figlio sinistro del nodo di sostituzione.
4) Impostate il puntatore al sottoalbero destro (sinistro) del nodo di sostituzione in modo che faccia riferimento al sottoalbero destro (sinistro) del nodo da eliminare.
5) Eliminate il nodo puntato dalla variabile temporanea di tipo puntatore.

Scrivete la funzione `deleteNode` che accetti come suoi argomenti un puntatore al nodo radice dell'albero e il valore da eliminare. La funzione dovrà individuare all'interno dell'albero il nodo contenente il valore da eliminare, e usare gli algoritmi discussi in questo esercizio per rimuovere il nodo. Qualora il valore non sia stato ritrovato nell'albero, la funzione dovrà visualizzare un messaggio che indichi se il valore sia stato eliminato o no. Modificate il programma della Figura 12.19, in modo che usi questa funzione. Dopo aver eliminato un elemento, richiamate le funzioni di visita `inOrder`, `preOrder` e `postOrder` per confermare che l'operazione di rimozione sia stata eseguita correttamente.

12.23 (*Ricerca in un albero binario*) Scrivete la funzione `binaryTreeSearch` che tenti di individuare un dato valore all'interno di un albero di ricerca binaria. La funzione dovrà accettare come argomenti un puntatore al nodo radice dell'albero e una chiave di ricerca da individuare. Qualora il nodo contenente la chiave di ricerca sia stato individuato, la funzione dovrà restituire un puntatore a quel nodo; in caso contrario, la funzione dovrà restituire un puntatore `NULL`.

12.24 (*Visita per livelli di un albero binario*) Il programma della Figura 12.19 ha mostrato tre metodi ricorsivi per visitare un albero binario: la visita simmetrica, quella anticipata e quella differita. Questo esercizio presenterà la visita per livelli di un albero binario, nella quale i valori dei nodi saranno visualizzati livello dopo livello, incominciando da quello del nodo radice. I nodi di ogni livello saranno visualizzati da sinistra a destra. La visita per livelli non è un algoritmo ricorsivo. Essa usa la struttura di dati coda per controllare l'output dei nodi. L'algoritmo sarà il seguente.

1) Inserite il nodo radice nella coda.
2) Fintanto che siano rimasti dei nodi nella coda,
   Prendete il nodo successivo della coda
   Visualizzate il valore del nodo
   Qualora il puntatore al figlio sinistro del nodo non sia il valore nullo
      Inserite il figlio sinistro nella coda
   Qualora il puntatore al figlio destro del nodo non sia il valore nullo
      Inserite il figlio destro nella coda

Scrivete la funzione `levelOrder` che esegua una visita per livelli di un albero binario. La funzione dovrà accettare come argomento un puntatore al nodo radice dell'albero. Modificate il programma della Figura 12.19, in modo che utilizzi questa funzione. Confrontate l'output di questa funzione con quelli degli altri algoritmi di visita per verificare che funzioni correttamente. [*Nota*: in questo programma, avrete anche bisogno di modificare e incorporare le funzioni per l'elaborazione delle code mostrate nella Figura 12.13.]

12.25 (*Visualizzare gli alberi*) Scrivete una funzione ricorsiva `outputTree` che visualizzi sullo schermo un albero binario. La funzione dovrà visualizzare l'albero una riga per volta, con la cima (la radice) posta alla sinistra dello schermo e la base (le foglie) sistemata a destra dello stesso. Ogni riga sarà visualizzata verticalmente. Per esempio, l'albero binario mostrato nella Figura 12.22 sarà visualizzato come segue:

```
                    99
               97
                    92
          83
                    72
               71
                    69
49
                    44
               40
                    32
          28
                    19
               18
                    11
```

Osservate che la foglia più a destra compare in cima all'output, nella colonna più a destra, mentre il nodo radice compare a sinistra dell'output. Ogni colonna dell'output incomincia cinque spazi più a destra di quella precedente. La funzione `outputTree` dovrà ricevere come argomenti un puntatore al nodo radice dell'albero e l'intero `totalSpaces`. Questa variabile indicherà il numero di spazi che dovranno precedere il valore da visualizzare e dovrà cominciare con zero, così che il nodo radice sia visualizzato a sinistra dello schermo. Per visualizzare l'albero, la funzione userà una versione modificata della visita simmetrica: essa incomincerà con il nodo più a destra dell'albero e proseguirà tornando indietro a sinistra. L'algoritmo sarà dunque il seguente:

Fintanto che il puntatore al nodo corrente non sia il valore nullo
Chiamate in modo ricorsivo `outputTree` con il sottoalbero destro del nodo corrente e `totalSpaces + 5`
Usate una struttura `for` per contare da 1 a `totalSpaces` e visualizzare gli spazi
Visualizzate il valore del nodo corrente
Impostate il puntatore al nodo corrente in modo che faccia riferimento al suo sottoalbero sinistro
Incrementate `totalSpaces` di 5.
Sezione speciale: costruite il vostro compilatore

Negli Esercizi 7.18 e 7.19, abbiamo introdotto il Linguaggio Macchina Simpletron (LMS) e creato il simulatore di computer Simpletron per eseguire i programmi scritti in LMS. In questa sezione, costruiremo un compilatore che convertirà in LMS dei programmi scritti in un linguaggio di programmazione di alto livello. Questa sezione "legherà" insieme l'intero processo di programmazione. Scriveremo dei programmi nel nuovo linguaggio di alto livello, li compileremo con il compilatore che avremo costruito e li eseguiremo con il simulatore che abbiamo costruito nell'Esercizio 7.19.

12.26 (*Il Linguaggio Semplice*) Prima di cominciare a costruire il compilatore, discuteremo un semplice ma potente linguaggio di alto livello simile alle prime versioni del popolare linguaggio BASIC. Chiameremo questo linguaggio Semplice. Ogni costrutto eseguibile di Semplice sarà formato da un numero di riga e da un'istruzione di Semplice. I numeri di riga dovranno comparire in ordine crescente. Ogni istruzione incomincerà con uno dei seguenti comandi di Semplice: `rem`, `input`, `let`, `print`, `goto`, `if...goto` o `end` (consultate la Figura 12.23). Eccettuato `end`, tutti i comandi potranno essere usati ripetutamente. Semplice valuterà soltanto delle espressioni intere usando gli operatori `+`, `-`, `*` e `/`.

Questi operatori manterranno la stessa priorità che hanno in C. Potranno comunque essere utilizzate delle parentesi per cambiare l'ordine di valutazione di un'espressione.

Il nostro compilatore di Semplice riconoscerà soltanto le lettere minuscole. Tutti i caratteri inclusi in un file di Semplice dovranno essere in minuscolo (quelli maiuscoli provocheranno un errore di sintassi, sempre che non compaiano in un'istruzione `rem`, nel qual caso saranno ignorati). I *nomi di variabile* saranno formati da una singola lettera. Il Semplice non consentirà nomi di variabile descrittivi, perciò queste dovranno essere descritte nei commenti in modo da indicare il loro uso all'interno del programma. Il Semplice userà soltanto delle variabili intere e non sarà necessario dichiararle: basterà menzionare all'interno del programma un nome di variabile, perché questa sia automaticamente dichiarata e inizializzata a zero. La sintassi di Semplice non consentirà la manipolazione delle stringhe (lettura, scrittura, confronto, ecc.). Il compilatore genererà un errore di sintassi, qualora incontri all'interno di un programma in linguaggio Semplice una stringa preceduta da un comando diverso da `rem`. Il nostro compilatore presumerà che i programmi in linguaggio Semplice siano stati immessi correttamente. L'Esercizio 12.29 chiederà al lettore di modificare il compilatore così che possa eseguire la verifica della sintassi.

Semplice userà l'istruzione condizionale `if...goto` e quella incondizionata `goto` per alterare il flusso di controllo durante l'esecuzione del programma. Qualora l'istruzione condizionata `if...goto` sia vera, il controllo sarà trasferito a una riga specifica del programma. In una istruzione `if...goto` saranno validi i seguenti operatori relazionali e di uguaglianza: `<`, `>`, `<=`, `>=`, `==` o `!=`. La priorità di questi operatori sarà la stessa che hanno in C.

Comando	Istruzione di esempio	Descrizione
`rem`	`50 rem questo è un commento`	Qualsiasi testo successivo al comando `rem` sarà inserito soltanto a scopo documentativo e sarà ignorato dal compilatore.
`input`	`30 input x`	Visualizzerà un punto interrogativo per chiedere all'utente di immettere un intero. Leggerà l'intero dalla tastiera e lo immagazzinerà in `x`.
`let`	`80 let u = 4 * (j - 56)`	Assegnerà a `u` il valore `4 * (j - 56)`. Notate che, a destra del segno di uguale, potrebbe comparire un'espressione arbitrariamente complessa.
`print`	`10 print w`	Visualizzerà il valore di `w`.
`goto`	`70 goto 45`	Trasferirà il controllo del programma alla riga `45`.
`if...goto`	`35 if i == z goto 80`	Confronterà `i` e `z` per verificarne l'uguaglianza e trasferirà il controllo del programma alla riga `80`, qualora la condizione risulti vera; in caso contrario, continuerà l'esecuzione con l'istruzione successiva.
`end`	`99 end`	Terminerà l'esecuzione del programma.

Figura 12.23 I comandi di Semplice

Prendiamo ora in considerazione alcuni programmi scritti in linguaggio Semplice che dimostreranno le caratteristiche di quest'ultimo. Il primo programma (Figura 12.24) leggerà due interi dalla tastiera, immagazzinerà i loro valori nelle variabili `a` e `b` e calcolerà e visualizzerà la loro somma (immagazzinata nella variabile `c`).

Il programma della Figura 12.25 determinerà e visualizzerà il maggiore tra due numeri interi. Gli interi saranno letti dalla tastiera e immagazzinati in `s` e `t`. L'istruzione `if...goto` verificherà la condizione `s >= t`. Qualora la condizione sia vera, il controllo sarà trasferito alla riga `90` e `s` sarà visualizzato; altrimenti, verrà visualizzato `t` e il controllo sarà trasferito all'istruzione `end` della riga `99`, dove il programma terminerà la propria esecuzione.

```
 1   10 rem   determinare e visualizzare la somma di due interi
 2   15 rem
 3   20 rem   prende in input i due interi
 4   30 input a
 5   40 input b
 6   45 rem
 7   50 rem   somma gli interi e immagazzina il risultato in c
 8   60 let c = a + b
 9   65 rem
10   70 rem visualizza il risultato
11   80 print c
12   90 rem   termina l'esecuzione del programma
13   99 end
```

Figura 12.24 Determinare la somma di due interi

```
 1   10 rem determinare il maggiore di due interi
 2   20 input s
 3   30 input t
 4   32 rem
 5   35 rem verifica se s >= t
 6   40 if s >= t goto 90
 7   45 rem
 8   50 rem t è maggiore di s, perciò visualizza t
 9   60 print t
10   70 goto 99
11   75 rem
12   80 rem s è maggiore o uguale a t, perciò visualizza s
13   90 print s
14   99 end
```

Figura 12.25 Trovare il maggiore di due interi

Il Semplice non fornisce una struttura di iterazione (come `for`, `while` o `do...while` del C). Tuttavia, può simulare ogni struttura di iterazione del C, usando le istruzioni `if...goto` e `goto`. La Figura 12.26 userà un ciclo controllato da un valore sentinella per calcolare i quadrati di vari interi. Ogni intero sarà preso in input dalla tastiera e immagazzinato nella variabile `j`. Qualora il valore immesso corrisponda a quello di guardia, `-9999`, il controllo sarà trasferito alla riga `99` dove l'esecuzione del programma terminerà. In caso contrario, il quadrato di `j` sarà assegnato a `k`, questa sarà visualizzata sullo schermo e il controllo sarà passato alla riga `20`, dove sarà preso in input l'intero successivo.

Usando come vostra guida i programmi di esempio delle Figure 12.24, 12.25 e 12.26, scrivete un programma in linguaggio Semplice che esegua ognuna delle seguenti attività:

a) Prendete in input tre interi, determinatene la media e visualizzate il risultato.

b) Usate un ciclo controllato da un valore sentinella per prendere in input 10 interi e calcolare e visualizzare la loro somma.
c) Usate un ciclo controllato da un contatore per prendere in input sette interi, alcuni positivi e altri negativi, e calcolare e visualizzare la loro media.
d) Prendete in input una serie di interi e determinatene e visualizzatene il maggiore. Il primo intero in input indicherà quanti numeri dovranno essere elaborati.
e) Prendete in input 10 interi e visualizzatene il minore.
f) Calcolate e visualizzate la somma degli interi pari compresi tra 2 e 30.
g) Calcolate e visualizzate il prodotto degli interi dispari compresi tra 1 e 9.

```
 1   10 rem   calcolare i quadrati di vari interi
 2   20 input j
 3   23 rem
 4   25 rem   controlla il valore sentinella
 5   30 if j == -9999 goto 99
 6   33 rem
 7   35 rem   calcola il quadrato di j e assegna il risultato a k
 8   40 let k = j * j
 9   50 print k
10   53 rem
11   55 rem   itera per prendere il prossimo j
12   60 goto 20
13   99 end
```

Figura 12.26 Calcolare il quadrato di vari interi

12.27 (*Costruire un compilatore*; prerequisiti: completate l'Esercizio 7.18, l'Esercizio 7.19, l'Esercizio 12.12, l'Esercizio 12.13 e l'Esercizio 12.26) Ora che il linguaggio Semplice è stato presentato (Esercizio 12.26), discuteremo di come costruire il nostro compilatore di Semplice. In primo luogo, considereremo il processo attraverso il quale un programma in linguaggio Semplice sarà convertito in LMS ed eseguito dal simulatore Simpletron (consultate la Figura 12.27). Un file contenente un programma in linguaggio Semplice sarà letto dal compilatore e convertito in codice LMS. Questo sarà salvato in un file su disco, nel quale le istruzioni LMS compariranno una per riga. Il file LMS sarà quindi caricato nel simulatore Simpletron e i risultati saranno inviati a un file su disco e allo schermo. Osservate che il programma Simpletron sviluppato nell'Esercizio 7.19 prendeva il suo input dalla tastiera. Esso dovrà quindi essere modificato per leggere l'input da un file, così che possa eseguire i programmi prodotti dal nostro compilatore.

Per convertirlo in LMS, il compilatore effettuerà due *passaggi* sul programma in linguaggio Semplice. Il primo passaggio costruirà la *tabella dei simboli* (la tabella dei simboli sarà discussa più tardi in dettaglio) nella quale saranno immagazzinati, con i loro tipi di dato e le rispettive posizioni all'interno

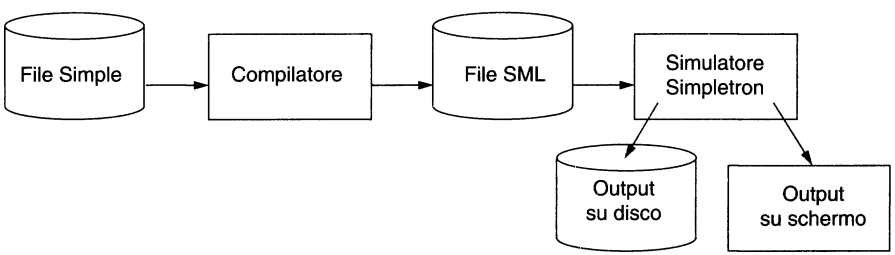


Figura 12.27 Scrivere, compilare ed eseguire un programma in linguaggio Semplice

del codice LMS finale, ogni *numero di riga, nome di variabile* e *costante* del programma in linguaggio Semplice. Il primo passo produrrà anche le istruzioni LMS corrispondenti a ogni espressione di Semplice. Come vedremo, il primo passaggio produrrà un codice LMS che conterrà alcune istruzioni incomplete, qualora il programma in linguaggio Semplice contenga delle istruzioni che trasferiscano il controllo a una riga successiva del codice. Il secondo passaggio del compilatore individuerà e completerà le istruzioni incomplete e salverà il programma LMS in un file.

Primo passaggio

Il compilatore inizierà leggendo nella memoria un'istruzione del programma in linguaggio Semplice. Per l'elaborazione e la compilazione, la riga dovrà essere separata nei suoi singoli *token* (ovverosia, nei "pezzi" di un'istruzione) e, per facilitare questo compito, potrà essere usata la funzione `strtok` della libreria standard. Ricordate che ogni istruzione incomincia con un numero di riga seguito da un comando. Man mano che il compilatore scomporrà un'istruzione nei suoi token, questi saranno sistemati nella tabella dei simboli, qualora siano dei numeri di riga, delle variabili o delle costanti. Un numero di riga sarà sistemato nella tabella dei simboli soltanto qualora sia il primo token di un'istruzione. La `symbolTable` è un vettore di strutture `tableEntry` che rappresenterà ogni simbolo del programma e, di conseguenza, potrà essere particolarmente corposa per certi programmi. Per ora, create la `symbolTable` come un vettore di 100 elementi. Potrete incrementare o ridurre le sue dimensioni, una volta che il programma sarà funzionante.

La definizione della struttura `tableEntry` è la seguente:

```
struct tableEntry {
   int symbol;
   char type;          /* 'C', 'L', o 'V' */
   int location;       /* da 00 a 99 */
}
```

Ogni struttura `tableEntry` contiene tre membri. Il membro `symbol` è un intero che conterrà la rappresentazione `ASCII` di una variabile (ricordate che i nomi di variabile sono caratteri singoli), un numero di riga o una costante. Il membro `type` corrisponderà a un carattere che indicherà il tipo del simbolo: `'C'` per una costante, `'L'` per un numero di riga o `'V'` per una variabile. Il membro `location` conterrà la posizione (da `00` a `99`) puntata dal simbolo nella memoria del Simpletron. Questa è un vettore di 100 elementi interi in cui saranno immagazzinati i dati e le istruzioni LMS. Per un numero di riga, la posizione rappresenterà l'elemento nel vettore della memoria del Simpletron in cui incominceranno le istruzioni LMS corrispondenti a quella in linguaggio Semplice. Per una variabile o una costante, la posizione rappresenterà l'elemento nel vettore della memoria del Simpletron in cui sarà immagazzinata la variabile, o la costante. Queste saranno allocate cominciando dalla fine della memoria del Simpletron. La prima variabile o costante sarà immagazzinata nella posizione `99`, la successiva nella `98` ecc.

La tabella dei simboli gioca una parte integrante nella conversione in LMS dei programmi scritti in linguaggio Semplice. Nel Capitolo 7 abbiamo appreso che un'istruzione LMS è un intero di quattro cifre, formato da due parti: il *codice dell'operazione* e l'*operando*. Il codice dell'operazione è determinato dai comandi del linguaggio Semplice. Per esempio, il comando di Semplice `input` corrisponde al codice dell'operazione LMS `10` (read), mentre il comando di Semplice `print` corrisponde al codice dell'operazione LMS `11` (write). L'operando è una posizione di memoria contenente i dati su cui il codice dell'operazione dovrà eseguire il proprio compito (il codice dell'operazione `10`, per esempio, leggerà un valore dalla tastiera e lo immagazzinerà nella posizione di memoria specificata dall'operando). Per ogni simbolo, il compilatore effettuerà una ricerca nella `symbolTable`, per determinarne la posizione all'interno della memoria del Simpletron, così che la stessa possa essere utilizzata per completare le istruzioni LMS.

La compilazione di ogni istruzione di Semplice è basata sui suoi comandi. Per esempio, una volta che il numero di riga di un `rem` sarà stato inserito nella tabella dei simboli, la parte rimanente dell'istruzione sarà ignorata dal compilatore, poiché un commento ha solo scopi documentativi. Le istruzioni `input`, `print`, `goto` ed `end` corrisponderanno a quelle LMS *read, write, branch* (a una posizione specificata) e *halt*. Le istruzioni contenenti questi comandi di Semplice saranno convertite direttamente in LMS [*Nota*: un `goto` potrebbe contenere un riferimento irrisolto, qualora il numero di riga punti a un'istruzione successiva nel file del programma in linguaggio Semplice; ovverosia il cosiddetto riferimento in avanti.]

Un `goto` con un riferimento irrisolto sarà compilato in un'istruzione LMS che dovrà essere *contrassegnata*, per indicare che il secondo passaggio del compilatore dovrà completarla. Le segnalazioni (o flag) saranno immagazzinate nei 100 elementi del vettore `flags` di tipo `int`, che saranno inizializzati con il valore `-1`. Nel caso in cui la posizione di memoria puntata da un numero di riga del programma in linguaggio Semplice non sia ancora nota (ovverosia, non sia stata ritrovata nella tabella dei simboli), allora quel numero di riga dovrà essere immagazzinato nell'elemento del vettore `flags` avente lo stesso indice dell'istruzione incompleta. L'operando dell'istruzione incompleta sarà impostato temporaneamente a `00`. Per esempio, un'istruzione di salto incondizionato (che abbia un riferimento in avanti) sarà lasciata come `+4000` fino al secondo passaggio del compilatore (che descriveremo tra breve).

La compilazione delle istruzioni `if...goto` e `let` è più complicata di quella delle altre: infatti, queste sono le uniche che producono più di un'istruzione LMS. Per un'istruzione `if...goto`, il compilatore produrrà un codice che verificherà la condizione e, se sarà necessario, salterà a un'altra riga. Il risultato del salto potrà anche essere un riferimento irrisolto. Ognuno degli operatori relazionali e di uguaglianza potrà essere simulato usando le istruzioni LMS *branch zero* e *branch negative*, o eventualmente una combinazione di entrambe.

Per un'istruzione `let`, il compilatore produrrà un codice che valuterà un'espressione aritmetica arbitrariamente complessa formata da variabili intere e/o costanti. Le espressioni dovranno separare ogni operando e operatore con degli spazi. Gli Esercizi 12.12 e 12.13 hanno presentato l'algoritmo di conversione dalla notazione infissa a quella polacca inversa, e l'algoritmo di valutazione usato dai compilatori per valutare le espressioni in notazione polacca inversa. Prima di procedere con il vostro compilatore, dovrete completare ognuno dei suddetti esercizi. Quando un compilatore incontrerà un'espressione, la convertirà dalla notazione infissa a quella polacca inversa e quindi la valuterà.

Ma in che modo il compilatore produrrà il codice per valutare un'espressione che contenga delle variabili? L'algoritmo di valutazione delle espressioni in notazione polacca inversa prevede una variante, che consentirà al nostro compilatore di generare le istruzioni LMS, invece di valutare effettivamente le espressioni. Per attivare la suddetta variante nel compilatore, l'algoritmo di valutazione delle espressioni in notazione polacca inversa dovrà essere modificato in modo che possa ricercare (ed eventualmente inserire) nella `symbolTable` ogni simbolo che incontrerà. Per ognuno di quei simboli, l'algoritmo dovrà quindi determinare e *inserire nella pila la corrispondente posizione di memoria, invece del simbolo*. Nel momento in cui verrà incontrato un operatore all'interno dell'espressione in notazione polacca inversa, saranno estratte le due posizioni di memoria in cima alla pila e sarà prodotto il codice in linguaggio macchina necessario per eseguire l'operazione, usando come operandi le posizioni di memoria. Il risultato di ogni componente dell'espressione sarà immagazzinato in una posizione di memoria temporanea e inserito nuovamente nella pila, perché possa proseguire la valutazione dell'espressione in notazione polacca inversa. Nel momento in cui la valutazione sarà stata completata, la posizione di memoria contenente il risultato sarà l'unica posizione rimasta sulla pila. Questa sarà dunque estratta e saranno generate le istruzioni LMS per assegnare il risultato alla variabile a sinistra dell'istruzione `let`.

Secondo passaggio

Il secondo passaggio del compilatore eseguirà due attività: risolvere ogni riferimento irrisolto e salvare il codice LMS in un file. La risoluzione dei riferimenti avverrà nel modo seguente:

1) Ricercate nel vettore `flags` un riferimento irrisolto (ovverosia, un elemento con un valore diverso da `-1`).
2) Individuate nel vettore `symbolTable` la struttura contenente il simbolo immagazzinato in `flags` (assicuratevi che il tipo del simbolo sia `'L'`, per i numeri di riga).
3) Inserite la posizione di memoria indicata dal membro della struttura `location` nell'istruzione con il riferimento irrisolto. Ricordatevi che un'istruzione contenente un riferimento irrisolto ha l'operando `00`.
4) Ripetete i passi 1, 2 e 3 finché non sarà stata raggiunta la fine del vettore `flags`.

Una volta che il processo di risoluzione sarà stato completato, l'intero vettore del codice LMS sarà salvato in un file su disco, sistemando un'istruzione LMS per riga. Questo file potrà essere letto dal Simpletron per l'esecuzione, una volta che avrete modificato il simulatore in modo che possa leggere il suo input da un file.

Un esempio completo

L'esempio seguente illustrerà una conversione completa in LMS di un programma scritto in linguaggio Semplice, così come sarà eseguita dal compilatore di Semplice. Considerate un programma scritto in linguaggio Semplice, che prenda in input un intero e sommi i valori compresi nell'intervallo da 1 all'intero specificato. Nella Figura 12.28 sono mostrati il programma e le istruzioni LMS generate dal primo passaggio. Nella Figura 12.29 è mostrata la tabella dei simboli costruita dal primo passaggio.

La maggior parte delle istruzioni di Semplice potrà essere convertita direttamente in singole istruzioni LMS. In questo programma, le eccezioni saranno rappresentate dai commenti, dall'istruzione `if...goto` della riga `20` e dall'istruzione `let`. I commenti non si traducono in linguaggio macchina. Tuttavia, i numeri di riga di un commento saranno ugualmente sistemati nella tabella dei simboli, nell'eventualità che siano stati puntati da un `goto` o da un `if...goto`. La riga `20` specifica che il controllo del programma passerà alla riga `60` qualora la condizione `y == x` sia vera. Dato che la riga `60` comparirà solo più tardi nel programma, il primo passaggio del compilatore non lo ha ancora sistemato nella tabella dei simboli (i numeri di riga saranno sistemati nella tabella dei simboli, solo qualora compaiano come primo token di un'istruzione). Di conseguenza, a questo punto, non è possibile determinare l'operando dell'istruzione LMS *branch zero*, nella posizione `03` del vettore che contiene le istruzioni LMS. Il compilatore sistemerà dunque il `60` nella posizione `03` del vettore `flags` per indicare che il secondo passaggio dovrà completare questa istruzione.

Programma in Semplice	Posizione e istruzione LMS	Descrizione
`5 rem aggiunge 1 a x`	*nessuna*	`rem` ignorata
`10 input x`	`00 +1099`	legge `x` nella posizione `99`
`15 rem verifica se y == x`	*nessuna*	`rem` ignorata
`20 if y == x goto 60`	`01 +2098`	carica `y (98)` nell'accumulatore
	`02 +3199`	sottrae `x (99)` dall'accumulatore
	`03 +4200`	*branch zero* a una posizione irrisolta

Figura 12.28 Le istruzioni LMS generate dopo il primo passaggio del compilatore (continua)

Programma in Semplice	Posizione e istruzione LMS	Descrizione
`25 rem incrementa y`	*nessuna*	`rem` ignorata
`30 let y = y + 1`	`04 +2098`	carica `y` nell'accumulatore
	`05 +3097`	aggiunge `1 (97)` all'accumulatore
	`06 +2196`	lo immagazzina nella posizione temporanea `96`
	`07 +2096`	carica dalla posizione temporanea `96`
	`08 +2198`	immagazzina l'accumulatore in `y`
`35 rem aggiunge y al totale`	*nessuna*	`rem` ignorata
`40 let t = t + y`	`09 +2095`	carica `t (95)` nell'accumulatore
	`10 +3098`	aggiunge `y` all'accumulatore
	`11 +2194`	immagazzina nella posizione temporanea `94`
	`12 +2094`	carica dalla posizione temporanea `94`
	`13 +2195`	immagazzina l'accumulatore in `t`
`45 rem ciclo y`	*nessuna*	`rem` ignorata
`50 goto 20`	`14 +4001`	salta alla posizione `01`
`55 rem visualizza il risultato`	*nessuna*	`rem` ignorata
`60 print t`	`15 +1195`	visualizza `t` sullo schermo
`99 end`	`16 +4300`	termina l'esecuzione

Figura 12.28 Le istruzioni LMS generate dopo il primo passaggio del compilatore

Simbolo	Tipo	Posizione
`5`	`L`	`00`
`10`	`L`	`00`
`'x'`	`V`	`99`
`15`	`L`	`01`
`20`	`L`	`01`
`'y'`	`V`	`98`
`25`	`L`	`04`
`30`	`L`	`04`
`1`	`C`	`97`
`35`	`L`	`09`
`40`	`L`	`09`

Figura 12.29 Tabella dei simboli per il programma della Figura 12.28 (continua)

Simbolo	Tipo	Posizione
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Figura 12.29 Tabella dei simboli per il programma della Figura 12.28

Dovremo mantenere traccia della posizione della prossima istruzione nel vettore LMS, perché non c'è una corrispondenza uno a uno tra le istruzioni di Semplice e quelle di LMS. Per esempio, l'istruzione `if...goto` della riga `20` sarà compilata in tre istruzioni LMS. Di conseguenza, ogni volta che sarà generata un'istruzione dovremo incrementare il *contatore di istruzioni*, in modo che punti alla posizione successiva del vettore LMS. Osservate che la dimensione della memoria del Simpletron potrebbe rappresentare un problema per i programmi in linguaggio Semplice formati da molte istruzioni, variabili e costanti. Infatti, è probabile che il compilatore esaurisca la memoria. Per controllare questa eventualità, il vostro programma dovrà contenere un *contatore di dati*, in modo da conservare la posizione del vettore LMS nella quale sarà immagazzinata la prossima variabile o costante. Il vettore LMS sarà pieno quando il valore del contatore di istruzioni sarà superiore a quello del contatore di dati. In questo caso, il processo di compilazione dovrà essere interrotto e il compilatore dovrà visualizzare un messaggio di errore, per indicare che avrà esaurito la memoria durante la compilazione.

Un esame passo per passo del processo di compilazione

Osserviamo ora il processo di compilazione per il programma in linguaggio Semplice mostrato nella Figura 12.28. Il compilatore legge in memoria la prima riga del programma

```
5 rem   aggiunge 1 a x
```

Il primo simbolo dell'istruzione, il numero di riga, sarà determinato usando `strtok` (consultate il Capitolo 8 per una discussione sulle funzioni C di manipolazione delle stringhe). Il simbolo restituito da `strtok` sarà convertito in un intero usando `atoi`, perciò nella tabella dei simboli potrà essere ricercato il token `5` che sarà inserito nella suddetta tabella, qualora non sia stato ritrovato. Dato che siamo all'inizio del programma e che questa è la prima riga, nella tabella non c'è ancora nessun simbolo. Di conseguenza, il `5` vi sarà inserito come tipo `L` (numero di riga) e sarà assegnato alla prima posizione (`00`) del vettore LMS. Nonostante si tratti di un commento, allocheremo ugualmente uno spazio per il numero di riga all'interno della tabella dei simboli, nell'eventualità che sia stato puntato da un `goto` o un `if...goto`. Per la `rem` non sarà generata nessuna istruzione LMS e quindi il relativo contatore non sarà incrementato.

In seguito, sarà analizzata l'istruzione

```
10 input x
```

Il numero di riga `10` sarà sistemato nella tabella dei simboli, come tipo `L`, e sarà assegnato alla prima posizione del vettore LMS (`00`, poiché il programma comincia con un commento e quindi il contatore di istruzioni varrà ancora `00`). Il comando `input` indica che il prossimo simbolo sarà una variabile, poiché in un'istruzione `input` può comparire solo una variabile. Il compilatore dovrà semplicemente determinare la posizione di `x` nel vettore LMS, poiché `input` corrisponde direttamente a un codice di operazione LMS. Il simbolo `x` non sarà ancora presente nella tabella dei simboli e quindi vi dovrà essere inserita la sua rappresentazione `ASCII`, con il tipo `V`, e gli sarà assegnata la posizione `99` del

vettore LMS, perché l'immagazzinamento dei dati incomincia dalla posizione 99 e prosegue dall'alto verso il basso. A questo punto potrà essere generato il codice LMS per questa istruzione. Il codice di operazione 10 (ovverosia quello LMS per la lettura) sarà moltiplicato per 100 e la posizione di x, determinata dalla tabella dei simboli, vi sarà aggiunta per completare l'istruzione. L'istruzione sarà quindi immagazzinata nella posizione 00 del vettore LMS. Il contatore di istruzioni sarà incrementato di 1 poiché sarà stata generata una sola istruzione LMS.

In seguito sarà analizzata l'istruzione

```
15 rem   verifica se y == x
```

Il numero di riga 15 sarà ricercato e non trovato nella tabella dei simboli, dove sarà inserito con il tipo L, e sarà assegnato alla posizione successiva del vettore, 01. Ricordate che l'istruzione rem non genera codice, perciò il contatore di istruzioni non sarà incrementato.

In seguito sarà analizzata l'istruzione

```
20 if y == x goto 60
```

Il numero di riga 20 sarà inserito nella tabella dei simboli e gli sarà assegnato il tipo L e la posizione successiva del vettore LMS: 01. Il comando if indica che dovrà essere valutata una condizione. La variabile y non sarà ancora presente nella tabella dei simboli, perciò vi sarà inserita e gli sarà assegnato il tipo V e la posizione 98 di LMS. In seguito, saranno generate le istruzioni LMS che valuteranno la condizione. Dato che in LMS non c'è nessuna diretta corrispondenza per il costrutto if/goto, questo dovrà essere simulato eseguendo un calcolo con x e y e trasferendo il controllo del programma in base al risultato. Il risultato della sottrazione di x da y sarà zero, qualora y sia uguale a x, e perciò potrà essere usata l'istruzione *branch zero* con il risultato del calcolo per simulare l'istruzione if/goto. Il primo passo richiederà che y sia caricata nell'accumulatore dalla posizione 99 di LMS, generando quindi l'istruzione 01 +2098. In seguito, x sarà sottratta dall'accumulatore, generando l'istruzione 02 +3199. A questo punto, il valore contenuto nell'accumulatore potrà essere zero, positivo o negativo.

Dato che l'operatore è ==, dovremo usare l'istruzione LMS *branch zero*. In primo luogo, cercheremo la posizione di destinazione per il salto (60, in questo caso) nella tabella dei simboli, ma non la troveremo. Di conseguenza, il 60 dovrà essere immesso nella posizione 03 del vettore flags e dovrà essere generata l'istruzione 03 +4200. Infatti, non potremo aggiungere la destinazione del salto, perché non avremo ancora assegnato una posizione nel vettore LMS alla riga 60. Il contatore di istruzioni sarà incrementato a 04.

Il compilatore procederà con l'istruzione

```
25 rem   incrementa y
```

Il numero di riga 25 sarà inserito nella tabella dei simboli con il tipo L e con la posizione 04 di LMS. Il contatore di istruzioni non sarà incrementato.

Nel momento in cui l'istruzione

```
30 let y = y +1
```

sarà analizzata, il numero di riga 30 sarà inserito nella tabella dei simboli con il tipo L e con la posizione 04 di LMS. Il comando let indica che la riga è un'istruzione di assegnamento e, di conseguenza, tutti i simboli della riga dovranno essere inseriti nella tabella dei simboli, qualora non vi siano già stati inclusi. L'intero 1 sarà aggiunto alla tabella dei simboli con il tipo C e con la posizione 97 di LMS. La parte destra dell'assegnamento sarà quindi convertita dalla notazione infissa a quella polacca inversa. In seguito, sarà valutata l'espressione in notazione polacca inversa (y 1 +). Il token y sarà individuato nella tabella dei simboli e la sua posizione di memoria sarà sistemata in cima alla pila. Anche il token 1 sarà individuato nella tabella dei simboli e la sua posizione di memoria sarà sistemata in cima alla pila. Una volta che avrà incontrato l'operatore +, la funzione di valutazione delle espressioni in notazione polacca inversa estrarrà due valori consecutivi dalla pila, assegnandoli rispettivamente agli operandi di destra e di sinistra dell'operatore e generando le istruzioni LMS

```
04 +2098      (load y)
05 +3097      (add 1)
```

Il risultato dell'espressione sarà immagazzinato in una posizione temporanea della memoria (96), con l'istruzione

```
06 +2196      (store temporaneo)
```

e la posizione temporanea sarà sistemata sulla pila. Una volta che l'espressione sarà stata valutata, il risultato dovrà essere immagazzinato in y (ovverosia, nella variabile a sinistra di =). Di conseguenza, la posizione temporanea sarà caricata nell'accumulatore e questo sarà immagazzinato in y, con le istruzioni

```
07 +2096      (load temporaneo)
08 +2198      (store y)
```

Vi sarete accorti immediatamente che le istruzioni LMS sembrano essere ridondanti. Discuteremo di questo tra breve.

Nel momento in cui l'istruzione

```
35 rem        aggiunge y al totale
```

sarà analizzata, il numero di riga 35 sarà inserito nella tabella dei simboli con il tipo L e con la posizione 09.

L'istruzione

```
40 let t = t + y
```

è simile alla riga 30. La variabile t sarà dunque inserita nella tabella dei simboli con il tipo V e con la posizione 95 di LMS. Le istruzioni seguono la stessa logica e formato della riga 30, perciò saranno generate le istruzioni 09 +2095, 10 +3098, 11 +2194, 12 +2094, e 13 +2195. Osservate che il risultato di t + y sarà assegnato alla posizione temporanea 94, prima di essere assegnato a t (95). Ancora una volta, avrete notato che le istruzioni inserite nelle posizioni di memoria 11 e 12 sembrano essere ridondanti. Di nuovo, discuteremo di ciò tra breve.

L'istruzione

```
45 rem   ciclo y
```

è un commento, perciò la riga 45 sarà aggiunta alla tabella dei simboli con il tipo L e con la posizione 14 di LMS.

L'istruzione

```
50 goto 20
```

trasferisce il controllo alla riga 20. In LMS l'equivalente di goto è l'istruzione di *salto incondizionato* (40) che trasferisce il controllo a una specifica posizione di LMS. Il compilatore cercherà dunque nella tabella dei simboli la riga 20 e rileverà che corrisponderà alla posizione 01 di LMS. Il codice dell'operazione (40) sarà moltiplicato per 100 e aggiunto alla posizione 01 così da generare l'istruzione 14 +4001.

L'istruzione

```
55 rem   visualizza il risultato
```

è un commento e, di conseguenza, la riga 55 sarà inserita nella tabella dei simboli con il tipo L e con la posizione 15.

L'istruzione

```
60 print t
```

è un'istruzione di output. Il numero di riga 60 sarà inserito nella tabella dei simboli con il tipo L e con la posizione 15 di LMS. In LMS l'equivalente di print è il codice di operazione 11 (*write*). La posizione di t sarà rilevata dalla tabella dei simboli e sarà aggiunta al prodotto tra 100 e il codice dell'operazione.

L'istruzione

```
99 end
```

è la riga finale del programma. Il numero di riga `99` sarà immagazzinato nella tabella dei simboli con il tipo `L` e con la posizione `16` di LMS. Il comando `end` genererà l'istruzione `+4300` (`43` è *halt* in LMS), che sarà scritta come istruzione finale nel vettore della memoria LMS.

Questo completa il primo passaggio del compilatore. Considereremo ora il secondo passaggio. Cercheremo nel vettore `flags` i valori diversi da `-1`. La posizione `03` contiene `60`, perciò il compilatore saprà che l'istruzione `03` non è completa. Esso integrerà quindi l'istruzione ricercando il `60` nella tabella dei simboli, determinandone la sua posizione e aggiungendola a quella incompleta. In questo caso, la ricerca determinerà che la riga `60` corrisponde alla posizione `15` di LMS e, di conseguenza, sarà generata l'istruzione completa `03 +4215` in sostituzione di `03 +4200`. A questo punto il programma scritto in Semplice sarà stato compilato con successo.

Per costruire il compilatore, dovrete eseguire ognuna delle seguenti attività:

a) Modificate il simulatore di Simpletron che avete scritto nell'Esercizio 7.19, in modo che prenda il suo input da un file specificato dall'utente (consultate il Capitolo 11). Il compilatore dovrà anche salvare i suoi risultati in un file su disco, seguendo lo stesso formato dell'output sullo schermo.

b) Modificate l'algoritmo di conversione dalla notazione infissa a quella polacca inversa dell'Esercizio 12.12, in modo che elabori operandi interi con più cifre e operandi corrispondenti a nomi di variabile formati da una sola lettera. [*Consiglio*: per individuare tutte le costanti e le variabili di un'espressione, potrà essere usata la funzione `strtok` della libreria standard, mentre le costanti di stringa potranno essere convertite in interi, usando la funzione `atoi` della libreria standard.] [*Nota*: la rappresentazione dei dati dell'espressione in notazione polacca inversa dovrà essere alterata in modo da supportare i nomi di variabile e le costanti intere.]

c) Modificate l'algoritmo di valutazione delle espressioni in notazione polacca inversa, in modo che elabori operandi interi con più cifre e operandi corrispondenti a nomi di variabile. L'algoritmo dovrà anche implementare la variante discussa in precedenza, in modo che siano prodotte le istruzioni LMS, invece di valutare direttamente l'espressione. [*Consiglio*: per individuare tutte le costanti e le variabili di un'espressione, potrà essere usata la funzione `strtok` della libreria standard, mentre le costanti di stringa potranno essere convertite in interi usando la funzione `atoi` della libreria standard.] [*Nota*: la rappresentazione dei dati dell'espressione in notazione polacca inversa dovrà essere alterata in modo da supportare i nomi di variabile e le costanti intere.]

d) Costruite il compilatore. Incorporate le parti (b) e (c) che valutino le espressioni delle istruzioni `let`. Il vostro programma dovrà contenere due funzioni che eseguano il primo e il secondo passo del compilatore. Entrambe le funzioni potranno richiamarne altre per svolgere i loro compiti.

12.28 (*Ottimizzare il compilatore di Semplice*) Quando un programma sarà compilato e convertito in LMS, sarà generato un insieme di istruzioni. Certe combinazioni di istruzioni si ripeteranno spesso, di solito in triplette dette produzioni. Queste saranno normalmente formate da tre istruzioni come load, add e store. Per esempio, la Figura 12.30 mostra cinque delle istruzioni LMS che sono state generate durante la compilazione del programma nella Figura 12.28. Le prime tre istruzioni corrispondono alla produzione che aggiunge `1` a `y`. Osservate che le istruzioni `06` e `07` immagazzinano il valore dell'accumulatore nella posizione temporanea `96` e poi lo ricaricano nuovamente nell'accumulatore, in modo che l'istruzione `08` possa immagazzinarlo nella posizione `98`. Spesso una produzione è seguita da un'istruzione di caricamento dalla stessa posizione che è stata appena usata per l'immagazzinamento. Questo codice potrà dunque essere ottimizzato, eliminando l'istruzione di immagazzinamento e quella susseguente di caricamento che operano sulla stessa posizione di memoria. L'ottimizzazione consentirà al

Simpletron di eseguire più velocemente il programma, poiché in questa versione ci saranno meno istruzioni. La Figura 12.31 mostra il codice LMS ottimizzato per il programma della Figura 12.28. Osservate che nel codice ottimizzato ci sono quattro istruzioni in meno: un risparmio di memoria del 25%.

```
04 +2098    (load)
05 +3097    (add)
06 +2196    (store)
07 +2096    (load)
08 +2198    (store)
```

Figura 12.30 Codice non ottimizzato tratto dal programma in Figura 12.28

Modificate il compilatore in modo da offrire un'opzione di ottimizzazione del codice generato in Linguaggio Macchina Simpletron. Confrontate manualmente il codice non ottimizzato e quello ottimizzato e calcolate la percentuale di riduzione.

Programma in Semplice	Posizione e istruzione LMS	Descrizione
`5 rem aggiunge 1 a x`	*nessuna*	`rem` ignorata
`10 input x`	`00 +1099`	legge `x` nella posizione `99`
`15 rem verifica se y == x`	*nessuna*	`rem` ignorata
`20 if y == x goto 60`	`01 +2098`	carica `y (98)` nell'accumulatore
	`02 +3199`	sottrae `x (99)` dall'accumulatore
	`03 +4211`	se zero, salta alla posizione `11`
`25 rem incrementa y`	*nessuna*	`rem` ignorata
`30 let y = y + 1`	`04 +2098`	carica `y` nell'accumulatore
	`05 +3097`	aggiunge `1 (97)` all'accumulatore
	`06 +2198`	immagazzina l'accumulatore in `y (98)`
`35 rem aggiunge y al totale`	*nessuna*	`rem` ignorata
`40 let t = t + y`	`07 +2096`	carica `t` dalla locazione `(96)`
	`08 +3098`	aggiunge `y (98)` all'accumulatore
	`09 +2196`	immagazzina l'accumulatore in `t (96)`
`45 rem ciclo y`	*nessuna*	`rem` ignorata
`50 goto 20`	`10 +4001`	salta alla posizione `01`
`55 rem visualizza il risultato`	*nessuna*	`rem` ignorata
`60 print t`	`11 +1196`	visualizza `t (96)` sullo schermo
`99 end`	`12 +4300`	termina l'esecuzione

Figura 12.31 Il codice ottimizzato per il programma della Figura 12.28

12.29 *(Modifiche al compilatore di Semplice)* Apportate le seguenti modifiche al compilatore di Semplice. Alcune di esse potrebbero anche richiedere delle variazioni al Simulatore Simpletron scritto nell'Esercizio 7.19.

a) Consentite l'uso dell'operatore modulo (`%`) nelle istruzioni `let`. Il Linguaggio Macchina Simpletron dovrà essere modificato per includere l'istruzione modulo.

b) Consentite l'elevamento a potenza nelle istruzioni `let`, usando il simbolo `^` come operatore di elevamento a potenza. Il Linguaggio Macchina Simpletron dovrà essere modificato per includere l'istruzione di elevamento a potenza.

c) Consentite al compilatore di riconoscere le lettere maiuscole e minuscole nelle istruzioni di Semplice (per esempio, `'A'` è equivalente ad `'a'`). Non saranno necessarie modifiche al Simulatore Simpletron.

d) Consentite all'istruzione `input` di leggere valori per variabili multiple come `input x, y`. Non saranno necessarie modifiche al Simulatore Simpletron.

e) Consentite al compilatore di visualizzare valori multipli in una singola istruzione `print` come `print a, b, c`. Non saranno necessarie modifiche al Simulatore Simpletron.

f) Aggiungete al compilatore la capacità di verificare la sintassi, in modo che siano visualizzati dei messaggi, qualora fossero riscontrati degli errori di sintassi all'interno di un programma in linguaggio Semplice. Non saranno necessarie modifiche al Simulatore Simpletron.

g) Implementate i vettori di interi. Non saranno necessarie modifiche al Simulatore Simpletron.

i) Implementate le subroutine specificate dai comandi di Semplice `gosub` e `return`. Il comando `gosub` passerà il controllo del programma a una subroutine, mentre il comando `return` lo restituirà all'istruzione successiva alla `gosub`. Questo meccanismo è simile alla chiamata di funzione del C. Una stessa subroutine potrà essere richiamata da molte `gosub` distribuite in tutto il programma. Non saranno richieste modifiche al Simulatore Simpletron.

j) Implementate le strutture di iterazione secondo il formato

```
for x = 2 to 10 step 2
      rem istruzioni in linguaggio Semplice
next
```

L'istruzione `for` itera da `2` a `10` con un incremento di `2`. La riga `next` indica la fine del corpo del `for`. Non saranno richieste modifiche al Simulatore Simpletron.

k) Implementate le strutture di iterazione secondo il formato

```
for x = 2 to 10
      rem istruzioni in linguaggio Semplice
next
```

L'istruzione `for` itera da `2` a `10` con un incremento predefinito di `1`. Non saranno richieste modifiche al Simulatore Simpletron.

l) Consentite al compilatore di elaborare l'input e l'output delle stringhe. Ciò richiederà la modifica del Simulatore Simpletron per consentirgli di elaborare e immagazzinare i valori di tipo stringa. [*Consiglio*: ogni parola del Simpletron potrà essere divisa in due gruppi che contengano un intero di due cifre. Ogni intero di due cifre rappresenterà il valore decimale ASCII equivalente a un carattere.] Aggiungete un'istruzione in linguaggio macchina che possa visualizzare una stringa cominciando da una certa posizione della memoria del Simpletron. La prima metà della parola in quella posizione corrisponderà al numero di caratteri presenti nella stringa (ovverosia, alla sua lunghezza). Ogni successiva mezza parola conterrà un carattere ASCII espresso con due cifre decimali. L'istruzione in linguaggio macchina controllerà la lunghezza e visualizzerà la stringa, traducendo ogni numero di due cifre nel carattere corrispondente.

m) Consentite al compilatore di elaborare i valori in virgola mobile, oltre a quelli interi. Anche il Simulatore Simpletron dovrà essere modificato per consentirgli di elaborare i valori in virgola mobile.

12.30 (*Un interprete di Semplice*) Un interprete è un programma che legge un'istruzione di un codice scritto in un linguaggio di alto livello, determina l'operazione da eseguire per quell'istruzione e la esegue immediatamente. Dunque il programma non sarà convertito prima in linguaggio macchina. Gli interpreti lavorano lentamente, perché ogni istruzione incontrata nel programma dovrà prima essere decifrata. Nel caso in cui le istruzioni siano contenute in un ciclo, queste saranno decifrate ogni volta che verranno incontrate all'interno del ciclo. Le prime versioni del linguaggio di programmazione BASIC furono implementate proprio come interpreti.

Scrivete un interprete per il linguaggio Semplice discusso nell'Esercizio 12.26. Il programma dovrà usare il convertitore da notazione infissa a polacca inversa, che avete sviluppato nell'Esercizio 12.12, e la funzione di valutazione di espressioni in notazione polacca inversa, che avete sviluppato nell'Esercizio 12.13, per valutare le espressioni di un'istruzione `let`. In questo programma, dovranno essere adottate le stesse restrizioni imposte dall'Esercizio 12.26 al linguaggio Semplice. Verificate il funzionamento dell'interprete con il programma in linguaggio Semplice scritto nell'Esercizio 12.26. Confrontate i risultati ottenuti dall'esecuzione interpretata di questo programma, con quelli ottenuti mediante la sua compilazione ed esecuzione con il simulatore Simpletron costruito nell'Esercizio 7.19.

Capitolo 13

Il preprocessore del C

Obiettivi

- Essere in grado di usare `#include` per sviluppare dei programmi corposi.
- Essere in grado di usare `#define` per creare delle macro con o senza argomenti.
- Comprendere la compilazione condizionale.
- Essere in grado di visualizzare i messaggi di errore durante la compilazione condizionale.
- Essere in grado di usare le asserzioni per verificare se i valori di un'espressione sono corretti.

13.1 Introduzione

Questo capitolo presenterà il *preprocessore del C* che interviene prima della compilazione di un programma. Alcune delle sue possibili azioni sono: l'inclusione di altri file in quello da compilare, la definizione delle *costanti simboliche* e delle *macro*, la *compilazione condizionale* del codice del programma e l'*esecuzione condizionale delle direttive del preprocessore*. Tutte le direttive del preprocessore incominciano con # e, prima delle suddette, in una riga possono esserci soltanto dei caratteri di spazio bianco.

13.2 La direttiva del preprocessore #include

La *direttiva del preprocessore* `#include` è stata usata dappertutto in questo libro. Essa consente di includere in sua vece la copia di un file specificato. Esistono due forme della direttiva `#include`:

```
#include <nome file>
#include "nome file"
```

La differenza tra queste due forme sta nella posizione in cui il preprocessore ricercherà il file da includere. Esso eseguirà la ricerca del file da includere nella stessa directory di quello da compilare, qualora il nome del file sia racchiuso tra virgolette. Questo metodo è usato normalmente per includere i file di intestazione definiti dal programmatore. La suddetta ricerca sarà invece eseguita in un modo dipendente dall'implementazione (normalmente in directory predefinite), qualora il nome del file sia racchiuso tra le parentesi ad angolo (< e >) che sono usate per i *file di intestazione della libreria standard.*

Normalmente la direttiva `#include` è usata per includere i file di intestazione della libreria standard come `stdio.h` e `stdlib.h` (consultate la Figura 5.6). La direttiva `#include` è

usata anche con quei programmi formati da vari file sorgente che dovranno essere compilati insieme. Spesso sarà creato e incluso nel sorgente un *file di intestazione* (o header file) che contiene le dichiarazioni comuni ai singoli file di un programma. Esempi di simili dichiarazioni sono quelle delle strutture e delle unioni, le enumerazioni e i prototipi di funzione.

13.3 La direttiva del preprocessore #define: le costanti simboliche

La *direttiva* `#define` crea le *costanti simboliche* (costanti rappresentate da simboli) e le *macro* (operazioni definite come simboli). Il formato della direttiva `#define` è

`#define` *identificatore testo-di-sostituzione*

Dal momento in cui questa riga compare in un file, tutte le successive occorrenze dell'*identificatore* saranno sostituite automaticamente dal *testo-di-sostituzione*, prima che il programma sia compilato. Per esempio,

```
#define PI 3.14159
```

sostituirà tutte le occorrenze della costante simbolica `PI` con quella numerica `3.14159`. Le costanti simboliche consentono al programmatore di attribuire un nome a una costante e di usarlo nel resto del programma. Qualora si avesse la necessità di modificare quella costante in tutto il programma, si potrà farlo modificandola solamente all'interno della direttiva `#define`. Tutte le occorrenze della costante saranno modificate di conseguenza, quando il programma sarà compilato di nuovo. [*Nota*: La costante simbolica sarà sostituita da tutto ciò che si trovi a destra del suo nome.] Per esempio, `#define PI = 3.14159` farà in modo che il preprocessore sostituisca ogni occorrenza di `PI` con `= 3.14159`. Questa è la causa di molti subdoli errori logici e sintattici. Un altro errore è di definire una costante simbolica con un nuovo valore.

Buona abitudine 13.1

Usare nomi significativi per le costanti simboliche aiuterà a rendere maggiormente autoesplicativo il programma.

13.4 La direttiva del preprocessore #define: le macro

Una *macro* è un identificatore definito all'interno di una direttiva `#define` del preprocessore. Al pari delle costanti simboliche, l'*identificatore della macro* sarà rimpiazzato dal *testo-di-sostituzione,* prima che il programma sia compilato. Le macro possono essere definite con o senza *argomenti*. Una macro senza argomenti sarà elaborata come una costante simbolica. In una *macro con argomenti*, questi saranno prima rimpiazzati all'interno del testo di sostituzione e solo in seguito sarà *espansa* la macro: in altri termini, il testo di sostituzione rimpiazzerà la lista degli identificatori e degli argomenti all'interno del programma.

Considerate la seguente definizione di macro con un argomento per l'area di un cerchio:

```
#define CIRCLE_AREA(x) ( PI * (x) * (x) )
```

In qualsiasi posto del file appaia `CIRCLE_AREA(y)`, il valore di `y` sarà usato al posto di `x` all'interno del testo di sostituzione, la costante simbolica `PI` sarà rimpiazzata dal suo valore (definito in precedenza) e la macro sarà espansa all'interno del programma. Per esempio, l'istruzione

```
area = CIRCLE_AREA(4);
```

sarà espansa in

```
area = ( 3.14159 * (4) * (4) );
```

e il valore dell'espressione sarà calcolato e assegnato alla variabile `area` durante la compilazione. Le parentesi intorno a ogni `x` nel testo di sostituzione, forzeranno il corretto ordine di valutazione, qualora l'argomento della macro dovesse essere un'espressione. Per esempio, l'istruzione

```
area = CIRCLE_AREA(c + 2);
```

sarà espansa in

```
area = ( 3.14159 * (c + 2) * (c + 2) );
```

che sarà valutata correttamente, poiché le parentesi forzeranno il corretto ordine di valutazione. Qualora le parentesi fossero state omesse, l'espansione della macro sarebbe stata

```
area = 3.14159 * c + 2 * c + 2;
```

che sarebbe stata valutata scorrettamente come

```
area = ( 3.14159 * c ) + (2 * c) + 2;
```

a causa delle regole di priorità degli operatori.

Errore tipico 13.1

Dimenticare di racchiudere tra parentesi gli argomenti delle macro nel testo di sostituzione può portare a commettere errori logici.

La macro `CIRCLE_AREA` potrebbe anche essere definita come una funzione. Per esempio, `circleArea`

```
double circleArea(double x)
{
   return 3.14159 * x * x;
}
```

eseguirebbe lo stesso calcolo della macro `CIRCLE_AREA`, ma a `circleArea` sarebbe associato il costo di una chiamata di funzione. I vantaggi della macro `CIRCLE_AREA` sono che questa inserirà il proprio codice direttamente nel programma, evitando quindi il costo di una funzione, e che il programma resterà leggibile, poiché il calcolo di `CIRCLE_AREA` sarà definito separatamente e avrà un nome significativo. Lo svantaggio è che il suo argomento sarà valutato due volte.

Obiettivo efficienza 13.1

A volte, le macro potranno essere utilizzate per sostituire una chiamata di funzione con un codice in linea, prima della fase di esecuzione. Ciò eliminerà il costo di una chiamata di funzione.

Quella che segue è una definizione di macro con due argomenti, per l'area di un rettangolo:

```
#define RECTANGLE_AREA(x, y) ( (x) * (y) )
```

In qualsiasi posto del programma appaia `RECTANGLE_AREA(x, y)`, i valori di `x` e di `y` saranno sostituiti nel testo di sostituzione e la macro sarà espansa per rimpiazzare il proprio nome. Per esempio, l'istruzione

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

sarà espansa in

```
rectArea = ( (a + 4) * (b + 7) );
```

Il valore dell'espressione sarà valutato e sarà assegnato alla variabile `rectArea`.

Normalmente, il testo di sostituzione di una macro o di una costante simbolica è tutto quello che segue l'identificatore nella riga della direttiva `#define`. Dovrete immettere un backslash (\) alla fine della riga per indicare che il testo di sostituzione continuerà su quella successiva, qualora il testo di una macro o di una costante simbolica sia più lungo della parte restante della riga.

Le costanti simboliche e le macro potranno anche essere eliminate usando la *direttiva del preprocessore* `#undef`. Questa "dimentica" il nome di una costante simbolica o di una macro. La *visibilità* di una costante simbolica o di una macro si estenderà dalla sua definizione fino al punto in cui sarà stata dimenticata con `#undef`, o fino alla fine del file. Una volta dimenticato, un nome potrà essere ridefinito con `#define`.

A volte, le funzioni della libreria standard sono definite proprio come macro basate su altre funzioni della libreria. Una macro definita comunemente nel file di intestazione `stdio.h` è

```
#define getchar() getc(stdin)
```

La definizione della macro `getchar` usa la funzione `getc` per prelevare un carattere dallo stream dello standard input. Anche la funzione `putchar` del file di intestazione `stdio.h` e quelle di manipolazione dei caratteri incluse in `ctype.h` sono spesso implementate come macro. Osservate che le espressioni con effetti collaterali (ovverosia, quelle in cui i valori delle variabili siano modificati) non dovrebbero essere passate a una macro, perché i suoi argomenti potrebbero essere valutati più di una volta.

13.5 La compilazione condizionale

La *compilazione condizionale* consente al programmatore di controllare l'esecuzione delle direttive del preprocessore e la compilazione del codice del programma. Ognuna delle direttive condizionali del preprocessore sarà valutata come un'espressione costante intera. Le espressioni di *cast* (conversione esplicita del tipo di dato), `sizeof` e le costanti di enumerazione non possono essere valutate nelle direttive del preprocessore.

Il costrutto condizionale del preprocessore è piuttosto simile al comando di selezione `if`. Considerate il codice di preprocessore seguente:

```
#if !defined(NULL)
   #define NULL 0
#endif
```

Queste direttive determineranno se `NULL` sia stato definito. L'espressione `defined(NULL)` sarà valutata `1`, qualora `NULL` sia stato definito; `0`, in caso contrario. Nel caso in cui il risultato sia `0`, `!defined(NULL)` sarà valutata `1` e `NULL` dovrà essere definito. In caso contrario, la direttiva `#define` dovrà essere ignorata. Ogni costrutto `#if` termina con `#endif`. Le direttive `#ifdef` e `#ifndef` sono abbreviazioni per `#if defined(`*nome*`)` e `#if !defined(`*nome*`)`. Un costrutto condizionale del preprocessore formato da più parti potrà essere costruito, usando le direttive `#elif` (l'equivalente di `else if` in un comando `if`) e `#else` (l'equivalente di `else` in un comando `if`).

Durante lo sviluppo dei programmi, i programmatori trovano spesso di grande aiuto la possibilità di "commentare" larghe porzioni di codice, per evitare che siano compilate. Nel caso in cui il codice contenga dei commenti, `/*` e `*/` non potranno essere usati per questo scopo, ma il programmatore potrà comunque usare il seguente costrutto del preprocessore:

```
#if 0
      codice che non deve essere compilato
#endif
```

Per consentire la compilazione del codice, lo `0` del costrutto precedente dovrà essere rimpiazzato da un `1`.

La compilazione condizionale è usata comunemente come un aiuto durante la messa a punto del software (debugging). Molte implementazioni del C forniscono dei *debugger*, che mettono a disposizione delle caratteristiche molto più potenti della compilazione condizionale. Se un debugger non è disponibile, si usano spesso delle istruzioni `printf` per visualizzare i valori delle variabili e convalidare il flusso di controllo. Queste potranno essere racchiuse in direttive condizionali del preprocessore, in modo che siano compilate soltanto finché non sarà stata completata la messa a punto. Per esempio,

```
#ifdef DEBUG
   printf("Variabile x = %d\n", x);
#endif
```

farà sì che l'istruzione `printf` sia compilata con il programma, qualora la costante simbolica `DEBUG` sia stata definita (`#define DEBUG`) prima della direttiva `#ifdef DEBUG`. La direttiva `#define` potrà essere rimossa dal file sorgente, una volta che la messa a punto sarà stata completata, così che le istruzioni `printf` inserite per quello scopo siano ignorate durante la compilazione. Nei programmi più corposi, è preferibile definire in sezioni distinte del file sorgente le varie costanti simboliche che controlleranno la compilazione condizionale.

Errore tipico 13.2

Inserire delle istruzioni `printf` *compilate condizionatamente per la messa a punto, in posizioni in cui il C richieda generalmente una singola istruzione. In questi casi, l'istruzione compilata in modo condizionale dovrà essere inclusa in una composta, così che il flusso di controllo del programma non risulti alterato, quando il codice sarà compilato con le istruzioni per la messa a punto.*

13.6 Le direttive del preprocessore #error e #pragma

La *direttiva* `#error`

```
#error token
```

visualizza un messaggio, dipendente dall'implementazione, che includerà i *token* (simboli) specificati all'interno della direttiva. I token sono sequenze di caratteri separati da spazi. Per esempio,

```
#error 1 - Out of range error
```

contiene sei token. Su alcuni sistemi nel momento in cui sarà elaborata una direttiva `#error`, i suoi token saranno visualizzati come un messaggio di errore, l'elaborazione del preprocessore sarà interrotta e il programma non sarà compilato.

La *direttiva* `#pragma`

```
#pragma token
```

provoca un'azione definita dall'implementazione. Una direttiva `#pragma` non riconosciuta dall'implementazione sarà ignorata. Per ottenere maggiori informazioni su `#error` e `#pragma`, consultate la documentazione della vostra implementazione del C.

13.7 Gli operatori # e

Gli operatori # e ## del preprocessore sono disponibili soltanto nello standard C. L'operatore # fa sì che un token del testo di sostituzione sia convertito in una stringa racchiusa tra virgolette. Considerate la seguente definizione di macro:

```
#define HELLO(x) printf("Hello, " #x "\n");
```

Nel punto del programma in cui apparirà `HELLO(John)`, questo sarà espanso in

```
printf("Hello, " "John" "\n");
```

La stringa `"John"` rimpiazzerà `#x` nel testo di sostituzione. Le stringhe separate da spazi bianchi saranno concatenate durante l'elaborazione del preprocessore, perciò la precedente istruzione è equivalente a

```
printf("Hello, John\n");
```

Osservate che l'operatore # dovrà essere usato in una macro con argomenti, poiché l'operando di # punta a un argomento della stessa.

L'operatore ## concatena due token. Considerate la seguente definizione di macro:

```
#define TOKENCONCAT(x, y) x ## y
```

Nel punto del programma in cui apparirà `TOKENCONCAT`, i suoi argomenti saranno concatenati e usati per sostituire la macro. Per esempio, `TOKENCONCAT(O, K)` sarà sostituita nel programma da `OK`. L'operatore ## dovrà avere due operandi.

13.8 I numeri di riga

La *direttiva del preprocessore* `#line` consente di assegnare una nuova numerazione alle righe successive del codice sorgente, partendo da un valore costante intero specificato.

La direttiva

```
#line 100
```

farà partire da `100` la numerazione della riga successiva del codice sorgente. Nella direttiva `#line` può anche essere incluso un nome di file. La direttiva

```
#line 100 "file1.c"
```

indica che le righe successive del codice sorgente saranno numerate cominciando da `100` e che il nome del file utilizzato da tutti i messaggi del compilatore dovrà essere `"file1.c"`. La direttiva è usata, normalmente, per aiutare a rendere più significativi i messaggi prodotti dagli errori di sintassi e dai warning (avvertimenti) del compilatore. I numeri di riga non compariranno nel file sorgente.

13.9 Le costanti simboliche predefinite

L'ANSI C mette a disposizione delle *costanti simboliche predefinite* (Figura 13.1). Gli identificatori di ognuna delle costanti simboliche predefinite cominciano e terminano con *due* caratteri di sottolineatura. Questi identificatori, insieme a quello `defined` (usato nella Sezione 13.5), non potranno essere usati nelle direttive `#define` e `#undef`.

Costante simbolica	Spiegazione
`__LINE__`	Il numero di linea della riga corrente nel codice sorgente (una costante intera).
`__FILE__`	Il nome presunto del file sorgente (una stringa).
`__DATE__`	La data in cui il file sorgente è stato compilato (una stringa nel formato `"Mmm gg aaaa"` come in `"Gen 19 1991"`).
`__TIME__`	L'ora in cui il file sorgente è stato compilato (una stringa letterale nel formato `"hh:mm:ss"`).

Figura 13.1 Alcune costanti simboliche predefinite

13.10 Le asserzioni

La macro `assert`, definita nel file di intestazione `assert.h`, controlla il valore di un'espressione. Qualora il valore di quella sia `0` (falso), `assert` visualizzerà un messaggio di errore e richiamerà la funzione `abort` (della libreria di utilità generiche, `stdlib.h`), per terminare l'esecuzione del programma. Questo è un utile strumento di messa a punto del software, che consente di controllare se una variabile contiene un valore corretto. Per esempio, supponete che in un programma la variabile `x` non debba mai essere maggiore di `10`. L'asserzione potrà essere usata per controllare il valore di `x` e visualizzare un messaggio di errore, qualora il valore di `x` non sia corretto. L'istruzione dovrebbe essere:

```
assert(x <= 10);
```

Nel caso in cui `x` sia maggiore di `10` quando l'istruzione precedente verrà incontrata in un programma, sarà visualizzato un messaggio di errore contenente il numero di riga e il nome del file, e l'esecuzione del programma sarà interrotta. A questo punto, per cercare l'errore, il

programmatore potrà concentrarsi su questa zona del codice. Nel caso in cui sia stata definita la costante simbolica NDEBUG, le asserzioni successive saranno ignorate. Per cui dovrete inserire nel programma la riga

```
#define NDEBUG
```

quando le asserzioni non saranno più necessarie, piuttosto che cancellare manualmente ogni asserzione.

Esercizi di autovalutazione

13.1 Riempite gli spazi vuoti in ognuna delle seguenti righe:

a) Ogni direttiva del preprocessore deve cominciare con _____.
b) Il costrutto della compilazione condizionale può essere esteso, per controllare casi multipli, usando le direttive _____ e _____.
c) La direttiva _____ crea delle macro e delle costanti simboliche.
d) Soltanto dei caratteri di _____ possono comparire in una riga, prima di una direttiva del preprocessore.
e) La direttiva _____ elimina i nomi delle costanti simboliche e delle macro.
f) Le direttive _____ e _____ sono fornite come notazione abbreviata per `#if defined(`*nome*`)` e `#if !defined(`*nome*`)`.
g) La _____ consente al programmatore di controllare l'esecuzione delle direttive del preprocessore e la compilazione del codice del programma.
h) La macro _____ visualizza un messaggio e terminerà l'esecuzione del programma, qualora il valore dell'espressione valutata dalla macro sia zero.
i) La direttiva _____ inserisce un file in un altro.
j) L'operatore _____ concatena i suoi due argomenti.
k) L'operatore _____ converte il suo operando in una stringa.
l) Il carattere _____ indica che il testo di sostituzione di una costante simbolica o di una macro continuerà sulla riga successiva.
m) La direttiva _____ consente di assegnare una nuova numerazione alle righe successive del codice sorgente, partendo dal valore specificato.

13.2 Scrivete un programma che visualizzi i valori delle costanti simboliche predefinite elencate nella Figura 13.1

13.3 Scrivete una direttiva del preprocessore che svolga ognuno dei seguenti compiti:

a) Definite la costante simbolica `YES` in modo che abbia il valore `1`.
b) Definite la costante simbolica `NO` in modo che abbia il valore `0`.
c) Includete il file di intestazione `common.h` che si trova nella stessa directory di quello da compilare.
d) Assegnate una nuova numerazione alle rimanenti righe del file, incominciando con il numero `3000`.
e) Nel caso in cui sia già stata definita la costante simbolica `TRUE`, rimuovete la sua definizione e definitela nuovamente in modo che abbia il valore `1`. Non usate `#ifdef`.
f) Nel caso in cui sia già stata definita la costante simbolica `TRUE`, rimuovete la sua definizione, e definitela nuovamente in modo che abbia il valore `1`. Usate la direttiva del preprocessore `#ifdef`.
g) Nel caso in cui la costante simbolica `TRUE` non sia uguale a `0`, definite la costante simbolica `FALSE` in modo che abbia il valore `0`. In caso contrario, definite `FALSE` in modo che abbia il valore `1`.

h) Definite la macro `SQUARE_VOLUME` che calcoli il volume di un quadrato. La macro dovrà accettare un argomento.

Risposte agli esercizi di autovalutazione

13.1 a) #. b) `#elif`, `#else`. c) `#define`. d) spazi bianchi. e) `#undef`. f) `#ifdef`, `#ifndef`. g) Compilazione condizionale. h) `assert`. i) `#include`. j) ##. k) #. l) \. m) `#line`.

13.2

```
1  /* Visualizzare i valori delle macro predefinite */
2   #include <stdio.h>
3   int main()
4   {
5       printf("__LINE__ = %d\n", __LINE__);
6       printf("__FILE__ = %d\n", __FILE__);
7       printf("__DATE__ = %d\n", __DATE__);
8       printf("__TIME__ = %d\n", __TIME__);
9       return 0;
10  }
```

```
__LINE__ = 5
__FILE__ = macros.c
__DATE__ = Jan 5 2003
__TIME__ = 09:38:58
```

13.3 a) `#define YES 1`

b) `#define NO 0`

c) `#include "common.h"`

d) `#line 3000`

e)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```

f)
```
#ifdef(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```

g)
```
#if TRUE
   #define FALSE 0
#else
   #define FALSE 1
#endif
```

h) `#define SQUARE_VOLUME(x) (x) * (x) * (x)`

Esercizi

13.4 Scrivete un programma che definisca una macro con un argomento per calcolare il volume di una sfera. Il programma dovrà calcolare il volume per sfere di raggio compreso tra 1 e 10 e visualizzare i risultati in formato tabulare. La formula per il volume di una sfera è:

```
(4/3) * π * r3
```

dove assumiamo per p un valore di `3.14159`.

13.5 Scrivete un programma che produca l'output seguente:

```
The sum of x and y is 13
```

Il programma dovrà definire la macro SUM con due argomenti, x e y, e utilizzare SUM per produrre l'output.

13.6 Scrivete un programma che utilizzi la macro MINIMUM2, per determinare il più piccolo tra due valori numerici che saranno presi in input dalla tastiera.

13.7 Scrivete un programma che usi la macro MINIMUM3 per determinare il più piccolo fra tre valori numerici. Per determinare il numero più piccolo, la suddetta dovrà utilizzare la macro MINIMUM2 definita nell'Esercizio 13.6. Prendete i valori in input dalla tastiera.

13.8 Scrivete un programma che usi la macro PRINT per visualizzare un valore di tipo stringa.

13.9 Scrivete un programma che usi la macro PRINTARRAY per visualizzare un vettore di interi. La macro dovrà ricevere come argomenti il vettore e il numero dei suoi elementi.

13.10 Scrivete un programma che usi la macro SUMARRAY per sommare i valori di un vettore numerico. La macro dovrà ricevere come argomenti il vettore e il numero dei suoi elementi.

Capitolo 14

Argomenti avanzati

Obiettivi

- Essere in grado di redirezionare l'input della tastiera in modo che provenga da un file.
- Essere in grado di redirezionare l'output dello schermo in modo che sia salvato in un file.
- Essere in grado di scrivere funzioni che utilizzino elenchi variabili di argomenti.
- Essere in grado di elaborare gli argomenti della riga di comando.
- Essere in grado di assegnare specifici tipi di dato alle costanti numeriche.
- Essere in grado di usare i file temporanei.
- Essere in grado di elaborare eventi inattesi in un programma.
- Essere in grado di allocare dinamicamente la memoria dei vettori.
- Essere in grado di modificare la dimensione della memoria allocata precedentemente in modo dinamico.

14.1 Introduzione

Questo capitolo presenterà diversi argomenti avanzati che non sono generalmente affrontati nei corsi introduttivi. Molte delle capacità discusse qui sono specifiche di un particolare sistema operativo, specialmente di UNIX e di Windows.

14.2 Redirezionare l'input/output su sistemi UNIX e Windows

Normalmente l'input di un programma proviene dalla tastiera (lo standard input), mentre il suo output è visualizzato sullo schermo (lo standard output). Sulla maggior parte dei sistemi di computer, in particolare su UNIX e Windows, è possibile *redirezionare* l'input in modo che provenga da un file, invece che dalla tastiera, e redirezionare l'output in modo che sia riversato in un file, invece che sullo schermo. Entrambe le forme di redirezionamento possono essere eseguite senza usare le capacità di elaborazione dei file fornite dalla libreria standard.

Esistono molti modi di redirezionare l'input e l'output dalla riga di comando di UNIX. Considerate il file eseguibile `sum` che, finché non sarà stato impostato l'indicatore di fine file, prenderà in input uno per volta degli interi, conserverà il totale corrente dei valori e infine visualizzerà il risultato. Normalmente l'utente inserirebbe gli interi dalla tastiera e premereb-

be la combinazione di tasti per l'indicatore di fine file, per comunicare che non saranno digitati ulteriori valori. Con il redirezionamento dell'input, questo potrà essere immagazzinato in un file. Per esempio, qualora i dati siano immagazzinati nel file `input`, la riga di comando

```
$ sum < input
```

eseguirà il programma `sum`. Il *simbolo di redirezionamento dell'input* (<) indica che il programma dovrà utilizzare i dati del file `input`. Il redirezionamento dell'input su un sistema Windows è eseguito allo stesso modo.

Osservate che il simbolo $ è un prompt della riga di comando in UNIX (alcuni sistemi UNIX usano per il prompt il simbolo % oppure un altro simbolo). Gli studenti trovano spesso difficile capire che il redirezionamento è una funzione del sistema operativo e non un'altra caratteristica del C.

Il secondo metodo di redirezionamento dell'input è il *piping* (convogliare in tubazioni). Un *pipe* (¦) fa in modo che l'output di un programma sia redirezionato nell'input di un altro. Supponete che il programma `random` visualizzi una serie di valori interi casuali; l'output di `random` potrà essere "convogliato" direttamente al programma `sum` usando la riga di comando UNIX

```
$ random ¦ sum
```

Questa determinerà il calcolo della somma degli interi prodotti da `random`. Il piping può essere eseguito sia in UNIX che in Windows.

L'output di un programma può essere redirezionato a un file, usando il *simbolo di redirezionamento dell'output* (>) (in UNIX e in Windows è utilizzato lo stesso simbolo). Per esempio, per redirezionare l'output del programma `random` al file `out`, userete

```
$ random > out
```

L'output di un programma, infine, può essere accodato alla fine di un file già esistente usando il *simbolo di accodamento dell'output* (>>) (in UNIX e in Windows è utilizzato lo stesso simbolo). Per esempio, per accodare l'output del programma `random` al file `out` creato con la riga di comando precedente, userete

```
$ random >> out
```

14.3 Gli elenchi variabili di argomenti

È possibile creare delle funzioni che ricevono un numero non specificato di argomenti. In questo testo, la maggior parte dei programmi ha utilizzato la funzione della libreria standard `printf` che, come sapete, accetta un numero variabile di argomenti. La funzione `printf` deve almeno ricevere come suo primo argomento una stringa, ma potrebbe anche ricevere un qualsiasi numero di argomenti aggiuntivi. Il prototipo della funzione `printf` è

```
int printf(const char *format, ...);
```

L'ellissi (`...`) nel prototipo della funzione indica appunto che questa riceve un numero variabile di argomenti di ogni tipo. Osservate che l'ellissi deve essere posta sempre alla fine dell'elenco degli argomenti.

Le macro e le definizioni incluse nel *file di intestazione per gli argomenti variabili* `stdarg.h` (Figura 14.1) forniscono i mezzi necessari per costruire delle funzioni con *elenchi variabili di*

Identificatore	Spiegazione
`va_list`	Un tipo adatto a contenere le informazioni necessarie alle macro `va_start`, `va_arg` e `va_end`. Per accedere agli elementi di un elenco variabile di argomenti, dovrà essere dichiarato un oggetto di tipo `va_list`.
`va_start`	Una macro che sarà invocata prima che si possa accedere agli elementi di un elenco variabile di argomenti. La macro inizializzerà l'oggetto dichiarato con `va_list` in modo che possa essere usato dalle macro `va_arg` e `va_end`.
`va_arg`	Una macro che si espanderà in un'espressione del valore e del tipo corrispondente all'elemento successivo di un elenco variabile di argomenti. Ogni invocazione di `va_arg` modificherà l'oggetto dichiarato con `va_list` in modo che possa puntare al prossimo argomento dell'elenco.
`va_end`	Una macro che faciliterà un normale ritorno da una funzione il cui elenco variabile di argomenti è stato puntato dalla macro `va_start`.

Figura 14.1 I tipi di dato e le macro definiti nel file di intestazione stdarg.h

argomenti. La Figura 14.2 mostra la funzione `average` (riga 28) che riceverà un numero variabile di argomenti. Il primo argomento di `average` sarà sempre il numero dei valori di cui calcolare la media.

La funzione `average` (righe 20-44) userà tutte le definizioni e le macro del file di intestazione `stdarg.h`. L'oggetto `ap` di tipo `va_list` (riga 32) sarà usato dalle macro `va_start`, `va_arg` e `va_end` per elaborare l'elenco variabile degli argomenti della funzione `average`. Questa incomincerà invocando la macro `va_start` (riga 34) per inizializzare l'oggetto `ap` così che possa essere usato da `va_arg` e `va_end`. La macro riceverà due dati: l'oggetto `ap` e l'identificatore dell'argomento più a destra che preceda l'ellissi nell'elenco degli argomenti, in questo caso `i` (`va_start` userà `i` per determinare dove incomincerà l'elenco variabile degli argomenti). In seguito, la funzione `average` aggiungerà ripetutamente a `total` gli elementi dell'elenco variabile degli argomenti (righe 37-39). Il valore che dovrà essere aggiunto a `total` sarà recuperato dall'elenco degli argomenti invocando la macro `va_arg`. Questa riceverà due argomenti: l'oggetto `ap` e il tipo di dato per il valore atteso nell'elenco degli argomenti, `double` in questo caso. La macro restituirà il valore dell'argomento. La funzione `average` invocherà la macro `va_end` (riga 41), usando come argomento l'oggetto `ap`, per facilitare un normale ritorno da `average` a `main`. Infine, la media sarà calcolata e restituita a `main`.

Errore tipico 14.1

Posizionare un'ellissi in mezzo all'elenco degli argomenti di una funzione è un errore di sintassi. L'ellissi può essere inserita soltanto alla fine dell'elenco degli argomenti.

Il lettore potrebbe chiedersi in che modo le funzioni `printf` e `scanf` possano conoscere il tipo di dato da usare, in ogni invocazione della macro `va_arg`. La risposta è che `printf` e `scanf`, per determinare il tipo del prossimo argomento da elaborare, esaminano gli indicatori di conversione inseriti nella stringa per il controllo del formato.

```
1  /* Fig. 14.2: fig14_02.c
2     Usare gli elenchi variabili di argomenti */
3  #include <stdio.h>
4  #include <stdarg.h>
5
6  double average(int, ...); /* prototipo */
7
8  int main()
9  {
10    double w = 37.5;
11    double x = 22.5;
12    double y = 1.7;
13    double z = 10.2;
14
15    printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
16        "w = ", w, "x = ", x, "y = ", y, "z = ", z);
17    printf("%s%.3f\n%s%.3f\n%s%.3f\n",
18        "The average of w and x is ", average(2, w, x),19
          "The average of w, x, and y is ", average(3, w, x, y),
20        "The average of w, x, y, and z is ",
21        average(4, w, x, y, z));
22
23    return 0; /* indica che il programma è terminato con successo */
24
25 } /* fine della funzione main */
26
27 /* calcola la media */
28 double average(int i, ...)
29 {
30    double total = 0; /* inizializza il totale */
31    int j; /* contatore per selezionare gli argomenti */
32    va_list ap; /* memorizza l'informazione necessaria
                     a va_start e va_end */
33
34    va_start(ap, i); /* inizializza l'oggetto va_list */
35
36    /* elabora l'elenco variabile di argomenti */
37    for (j = 1; j <= i; j++) {
38       total += va_arg(ap, double);
39    } /* fine del ciclo for */
40
41    va_end(ap); /* ripulisce l'elenco variabile di argomenti */
42
43    return total / i; /* calcola la media */
44 } /* fine della funzione average */
```

Figura 14.2 Usare gli elenchi variabili di argomenti (continua)

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Figura 14.2 Usare gli elenchi variabili di argomenti

14.4 Usare gli argomenti della riga di comando

Su molti sistemi è possibile passare degli argomenti dalla riga di comando alla funzione `main`, includendo `int argc` e `char *argv[]` nell'elenco degli argomenti di `main`. Il parametro `argc` riceve il numero degli argomenti presenti sulla riga di comando. Il parametro `argv` è un vettore di stringhe nel quale saranno immagazzinati gli argomenti della riga di comando corrente. Tra gli usi comuni degli argomenti sulla riga di comando, ricordiamo la visualizzazione degli stessi e il passaggio di opzioni e nomi di file al programma.

Il programma della Figura 14.3 copierà un file in un altro, un carattere per volta. Supponiamo che il file eseguibile del programma si chiami `mycopy`. Una tipica riga di comando per il programma `mycopy` su un sistema UNIX sarà

```
$ mycopy input output
```

Questa riga di comando indica che il file `input` dovrà essere ricopiato nel file `output`.

```
1  /* Fig. 14.3: fig14_03.c
2     Usare gli argomenti della riga di comando */
3  #include <stdio.h>
4
5  int main(int argc, char *argv[])
6  {
7     FILE *inFilePtr; /* puntatore al file di input */
8     FILE *outFilePtr; /* puntatore al file di output */
9     int c;            /* c viene dichiarata per memorizzare
                           i caratteri letti in input */
10
11    /* controlla il numero di argomenti presenti sulla linea
         di comando */
12    if (argc != 3) {
13       printf("Usage: copy infile outfile\n");
14    } /* fine del blocco if */
15    else {
16
17       /* se il file di input può essere aperto */
18       if ((inFilePtr = fopen(argv[1], "r")) != NULL) {
```

Figura 14.3 Usare gli argomenti della riga di comando

```
19
20          /* se il file di output può essere aperto */
21          if ((outFilePtr = fopen(argv[2], "w")) != NULL) {
22
23             /* legge e scrive i caratteri */
24             while ((c = fgetc(inFilePtr)) != EOF) {
25                fputc(c, outFilePtr);
26             } /* fine del comando while */
27
28          } /* fine del blocco if */
29          else { /* il file di output non può essere aperto */
30             printf("File \"%s\" could not be opened\n", argv[2]);
31          } /* fine del blocco else */
32
33       } /* fine del blocco if */
34       else { /* il file di input non può essere aperto */
35          printf("File \"%s\" could not be opened\n", argv[1]);
36       } /* fine del blocco else */
37
38    } /* fine del blocco else */
39
40    return 0; /* indica che il programma è terminato con successo */
41
42 } /* fine della funzione main */
```

Figura 14.3 Usare gli argomenti della riga di comando

Con l'esecuzione del programma, se `argc` non è 3 (`mycopy` conta come primo argomento), il programma visualizzerà un messaggio di errore e terminerà la propria esecuzione. In caso contrario, il vettore `argv` conterrà le stringhe `"mycopy"`, `"input"` e `"output"`. Il secondo e il terzo argomento della riga di comando saranno usati dal programma come nomi di file. Questi saranno aperti utilizzando la funzione `fopen`. Nel caso in cui entrambi i file siano stati aperti con successo, i caratteri saranno letti da `input` e scritti nel file `output`, finché nel primo non verrà incontrato l'indicatore di fine file. A questo punto, terminerà l'esecuzione del programma. Il risultato sarà una copia esatta del file `input`. Per ottenere maggiori informazioni sugli argomenti della riga di comando, consultate i manuali del vostro sistema.

14.5 Note sulla compilazione di programmi formati da vari file sorgente

Come affermato precedentemente in questo testo, è possibile costruire dei programmi che consistono di molti file sorgente. Ci sono molti fattori da tenere in considerazione quando si creano dei programmi suddivisi in molti file. Per esempio, la definizione di una funzione deve essere interamente contenuta in un file: non può estendersi per due o più file.

Nel Capitolo 5, abbiamo introdotto i concetti di classe di memoria e visibilità. Abbiamo imparato che le variabili dichiarate all'esterno di ogni definizione di funzione appartengono,

per default, alla classe di memoria statica e sono chiamate variabili globali. Queste, dopo la loro dichiarazione, sono accessibili a ogni funzione definita nello stesso file. Le variabili globali sono accessibili anche alle funzioni definite in altri file. Tuttavia, per far questo dovranno essere dichiarate in ognuno di quelli in cui saranno utilizzate. Per esempio, se definiamo in un file la variabile globale intera `flag` e vi facciamo riferimento in un secondo file, questo dovrà contenere prima dell'uso della variabile la dichiarazione

```
extern int flag;
```

Questa dichiarazione utilizza l'indicatore di classe di memoria `extern` per indicare che la variabile `flag` è definita più avanti nello stesso file o in un altro. Il compilatore informa il linker che nel file compare il riferimento irrisolto alla variabile `flag`: il compilatore non sa dove questa sia stata definita, perciò lascerà che sia il linker a tentare di trovare la definizione di quella variabile. Nel caso in cui il linker non dovesse riuscire a trovare una definizione di `flag`, genererebbe un messaggio d'errore e non produrrebbe nessun file eseguibile. Se il linker individua una definizione globale appropriata, risolverà i riferimenti indicando dove si trovi `flag`.

Obiettivo efficienza 14.1

Le variabili globali migliorano le prestazioni, poiché tutte le funzioni possono accedervi direttamente ed è eliminato il peso del passaggio dei dati alle funzioni.

Ingegneria del software 14.1

Le variabili globali dovrebbero essere evitate, sempre che le prestazioni dell'applicazione non siano critiche, perché esse violano il principio del minimo privilegio.

Proprio come le dichiarazioni `extern` possono essere usate per rendere note le variabili globali agli altri file del programma, i prototipi possono estendere la visibilità di una funzione oltre il file in cui è stata definita (in un prototipo di funzione, non è necessario l'indicatore `extern`). Tutto ciò si ottiene, includendo il prototipo della funzione in ognuno dei file in cui quella è invocata e compilando insieme i file (consultate la Sezione 13.2). I prototipi indicano al compilatore che la funzione specificata sarà definita più tardi nello stesso file o in un altro. Ancora una volta, il compilatore non tenta di risolvere i riferimenti a una tale funzione: questo compito è lasciato al linker. Qualora quest'ultimo non possa localizzare una definizione di funzione appropriata, genererà un messaggio d'errore.

Considerate tutti i programmi che contengono la direttiva del preprocessore `#include <stdio.h>`, come un esempio di utilizzo dei prototipi per estendere la visibilità di una funzione. Questa direttiva includerà in un file i prototipi per funzioni come `printf` e `scanf`. Le altre funzioni all'interno del file potranno usare `printf` e `scanf` per svolgere i loro compiti. Le funzioni `printf` e `scanf` sono definite in altri file. Non avremo bisogno di sapere dove sono state definite. Nel nostro programma, staremo semplicemente riutilizzando quel codice. Il linker risolverà automaticamente i nostri riferimenti a quelle funzioni. Questo processo ci consentirà di utilizzare le funzioni della libreria standard.

Ingegneria del software 14.2

Creare programmi suddivisi in molti file sorgente facilita il riutilizzo del software e una buona progettazione dello stesso. Le funzioni potrebbero essere comuni a molte applicazioni. In questa evenienza, quelle funzioni dovrebbero essere immagazzinate nei propri file sorgente e ognuno di questi dovrebbe avere un corrispondente file di

intestazione, contenente i prototipi delle funzioni. Ciò consentirà ai programmatori che svilupperanno altre applicazioni di riutilizzare lo stesso codice, includendo l'appropriato file di intestazione e compilando le proprie applicazioni con il corrispondente file sorgente.

È possibile restringere la visibilità di una variabile o di una funzione globale al file in cui queste sono state definite. L'indicatore di classe di memoria `static`, quando applicato a una variabile o a una funzione globale, impedisce che queste possano essere utilizzate da funzioni che non siano state definite all'interno dello stesso file. Questo è detto appunto *collegamento interno*. Le variabili e le funzioni globali che nella loro definizione non sono precedute da **`static`** determinano invece un *collegamento esterno*: esse sono accessibili ad altri file, qualora questi contengano le appropriate dichiarazioni e/o prototipi di funzione.

La dichiarazione di variabile globale

```
static double pi = 3.14159;
```

creerà la variabile `pi` di tipo `double`, la inizializzerà con `3.14159` e indicherà che `pi` sarà nota soltanto alle funzioni dichiarate nel file in cui quest'ultima è definita.

L'indicatore `static` è utilizzato comunemente con le funzioni ausiliarie richiamate soltanto da quelle di un particolare file. Nel caso in cui una funzione non sia necessaria all'esterno di un particolare file, dovrebbe essere osservato il principio del minimo privilegio, utilizzando `static`. Nel caso che una funzione sia stata definita prima del suo utilizzo all'interno di un file, allora `static` potrà essere applicato direttamente alla sua definizione. In caso contrario, `static` dovrà essere applicato al prototipo della funzione.

Quando si costruiscono dei programmi di dimensioni ragguardevoli suddivisi in molti file sorgente, la compilazione del programma diventa tediosa, se sono stati effettuati dei piccoli cambiamenti a un file e l'intero programma dovrà essere compilato nuovamente. Molti sistemi forniscono dei programmi di utilità speciali che compilano nuovamente soltanto i file del programma che sono stati modificati. Sui sistemi UNIX, tale programma è chiamato *`make`*. Questo legge un file, chiamato *`makefile`*, che contiene le istruzioni per la compilazione e il linking del programma. Anche prodotti come il Borland C++ e il Microsoft Visual C++ forniscono dei programmi di utilità simili. Per maggiori informazioni sui programmi di utilità `make`, consultate il manuale del vostro strumento di sviluppo.

14.6 Chiusura dei programmi con exit e atexit

La libreria delle utilità generiche (`stdlib.h`) fornisce alcuni metodi per terminare l'esecuzione del programma, diversi da quello convenzionale del ritorno dalla funzione `main`. La funzione *`exit`* forza la chiusura di un programma come se questo fosse stato eseguito normalmente. La funzione è usata spesso per terminare l'esecuzione di un programma, quando si rileva un errore nell'input, o quando non può essere aperto un file che dovrebbe essere elaborato dal programma. La funzione *`atexit`* *registra* una funzione da invocare subito dopo la chiusura con successo del programma: ovverosia, quando questo avrà terminato la propria esecuzione, per aver raggiunto la fine della funzione `main` o perché è stata invocata `exit`.

La funzione `atexit` riceve come argomento un puntatore a una funzione (ovverosia, il nome della stessa). La funzione richiamata alla chiusura del programma non può avere argomenti né può restituire un valore. Possono essere registrate fino a 32 funzioni da eseguire alla chiusura del programma.

La funzione `exit` riceve un argomento. Questo è normalmente la costante simbolica *`EXIT_SUCCESS`* o *`EXIT_FAILURE`*. Nel caso in cui `exit` sia stata invocata con `EXIT_SUCCESS`, sarà restituito all'ambiente chiamante il valore definito dall'implementazione per la chiusura con successo. Nel caso in cui `exit` sia stata invocata con `EXIT_FAILURE`, sarà restituito il valore definito dall'implementazione per la chiusura con fallimento. Nel momento in cui verrà richiamata la funzione `exit`, tutte quelle registrate in precedenza con `atexit` saranno invocate in ordine inverso alle proprie registrazioni, saranno svuotati e chiusi tutti gli stream associati al programma e il controllo ritornerà all'ambiente ospite. In Figura 14.4 vengono messe alla prova le funzioni `exit` e `atexit`. Il programma richiederà all'utente di indicare se dovrà terminare la propria esecuzione con `exit` o raggiungendo la fine di `main`. Osservate che la funzione `print` sarà eseguita, in ogni caso, alla chiusura del programma.

```
1  /* Fig. 14.4: fig14_04.c
2     Usare le funzioni exit e atexit */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  void print(void); /* prototipo */
7
8  int main()
9  {
10    int answer; /* scelta del menu da parte dell'utente */
11
12    atexit(print);       /* registra la funzione print */
13    printf("Enter 1 to terminate program with function exit"
14        "\nEnter 2 to terminate program normally\n");
15    scanf("%d", &answer);
16
17    /* richiama exit se answer vale 1 */
18    if (answer == 1) {
19       printf("\nTerminating program with function exit\n");
20       exit(EXIT_SUCCESS);
21    } /* fine del comando if */
22
23    printf("\nTerminating program by reaching the end of main\n");
24
25    return 0; /* indica che il programma è terminato con successo */
26
27 } /* fine della funzione main */
28
29 /* visualizza un messaggio prima della terminazione */
30 void print(void)
```

Figura 14.4 Le funzioni exit e atexit (continua)

```
31   {
32      printf("Executing function print at program termination\n"
33             "Program terminated\n");
34   } /* fine della funzione print */
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

Figura 14.4 Le funzioni exit e atexit

14.7 Il qualificatore di tipo volatile

Nei Capitoli 6 e 7, abbiamo introdotto il qualificatore di tipo `const`. L'ANSI C fornisce anche il qualificatore di tipo `volatile` per escludere vari tipi di ottimizzazioni. Lo standard C indica che, qualora sia usato `volatile` per qualificare un tipo, la natura dell'accesso a un oggetto di quel tipo dipenderà dall'implementazione.

14.8 I suffissi per le costanti intere e in virgola mobile

Il C fornisce alcuni suffissi per specificare i tipi delle costanti intere e in virgola mobile. I suffissi per le costanti intere sono: `u` o `U` per un intero `unsigned`, `l` o `L` per un intero `long`, e `ul`, `lu`, `UL` o `LU` per un intero `unsigned long`. Le seguenti costanti sono rispettivamente di tipo `unsigned`, `long` e `unsigned long`:

```
174u
8358L
28373ul
```

Nel caso in cui una costante intera non abbia un suffisso, sarà scelto come suo tipo di dato il primo in grado di immagazzinare un valore di quella dimensione (prima `int`, poi `long int`, poi `unsigned long int`).

I suffissi per le costanti in virgola mobile sono: `f` o `F` per un `float`, `l` o `L` per un `long double`. Le seguenti costanti sono rispettivamente di tipo `long double` e `float`:

```
3.14159L
1.28f
```

Una costante in virgola mobile senza suffisso sarà considerata automaticamente di tipo `double`.

14.9 Ancora sui file

Il Capitolo 11 ha trattato delle caratteristiche disponibili per l'elaborazione dei file di testo con l'accesso sequenziale e quello casuale. Il C fornisce anche la possibilità di elaborare i file binari, ma alcuni sistemi di computer non li supportano. Qualora non siano supportati e un file sia aperto in modo binario (Figura 14.5), questo sarà elaborato come un file di testo. I file binari dovrebbero essere usati, invece di quelli di testo, soltanto nelle situazioni in cui sono richiesti da condizioni rigorose di velocità, memoria, e/o compatibilità. Altrimenti, per la loro intrinseca portabilità e per la possibilità di usare altri strumenti standard per esaminare e manipolare i dati contenuti, dovranno sempre essere preferiti i file di testo.

Modo	Descrizione
`rb`	Apre un file binario per la lettura.
`wb`	Crea un file binario per la scrittura. Nel caso in cui il file sia già esistente, ne eliminerà il contenuto corrente.
`ab`	Accoda: apre o crea un file binario per la scrittura alla fine dello stesso.
`rb+`	Apre un file binario per l'aggiornamento (lettura e scrittura).
`wb+`	Crea un file binario per l'aggiornamento. Nel caso in cui il file sia già esistente, ne eliminerà il contenuto corrente.
`ab+`	Accoda: apre o crea un file binario per l'aggiornamento; tutte le scritture saranno eseguite alla fine dello stesso.

Figura 14.5 Modi di apertura di un file binario

Obiettivo efficienza 14.2

Prendete in considerazione l'uso dei file binari, invece di quelli di testo, nelle applicazioni che richiedano un'elevata efficienza.

Obiettivo portabilità 14.1

Usate i file di testo quando scrivete programmi portabili.

In aggiunta alle funzioni per l'elaborazione dei file discusse nel Capitolo 11, la libreria standard fornisce anche la funzione `tmpfile` che apre un file temporaneo nel modo `"wb+"`. Per quanto questo modo di apertura sia destinato ai file binari, alcuni sistemi elaborano gli archivi temporanei come file di testo. Un file temporaneo esisterà, finché non sarà stato chiuso con `fclose` o fino al termine dell'esecuzione del programma.

In Figura 14.6 verranno sostituiti con degli spazi i caratteri di tabulazione presenti in un file. Il programma richiederà all'utente di immettere il nome del file che dovrà essere modificato. Nel caso in cui il file immesso dall'utente e quello temporaneo siano stati aperti con successo, il programma leggerà i caratteri dal file da modificare e li scriverà in quello tempo-

raneo. Nel caso in cui il carattere letto sia una tabulazione (`'\t'`), sarà sostituito da uno spazio e scritto nel file temporaneo. Nel momento in cui verrà raggiunta la fine dell'archivio da modificare, i rispettivi puntatori di file saranno riposizionati con la `rewind` all'inizio di ognuno di essi. In seguito, il file temporaneo sarà ricopiato un carattere per volta su quello originale. Il programma visualizzerà il file originale, man mano che ricopierà i caratteri in quello temporaneo, e visualizzerà il nuovo file, man mano che ricopierà i caratteri da quello temporaneo a quello originale, per confermare che siano stati scritti.

```
1  /* Fig. 14.6: fig14_06.c
2     Usare i file temporanei */
3  #include <stdio.h>
4
5  int main()
6  {
7     FILE *filePtr; / puntatore al file da modificare */
8     FILE tempFilePtr; /* puntatore al file temporaneo */
9     int c; /* definizione di c per contenere I caratteri
                  letti dal file */
10    char fileName[30]; /* creazione di un array di caratteri */
11
12    printf("This program changes tabs to spaces.\n"
13           "Enter a file to be modified: ");
14    scanf("%s", fileName);
15
16    /* fopen apre il file */
17    if ( ( filePtr = fopen(fileName, "r+") ) != NULL)
18
19       /* creazione di un file temporaneo */
20       if ( ( tempFilePtr = tmpfile() ) != NULL) {
21          printf("\nThe file before modification is:\n");
22
23          /* legge I caratteri dal file e li memorizza
                nel file temporaneo */
24          while ( ( c = getc(filePtr) ) != EOF) {
25             putchar(c);
26             putc(c == '\t' ? ' ': c, tempFilePtr);
27          } /* fine del comando while */
28
29          rewind(tempFilePtr);
30          rewind(filePtr);
31          printf("\n\nThe file after modification is:\n");
32
33          /* legge dal file temporaneo e scrive nel file originale */
34          while ( ( c = getc(tempFilePtr) ) != EOF) {
35             putchar(c);
36             putc(c, filePtr);
37          } /* fine del comando while */
```

Figura 14.6 File temporanei (continua)

```
38
39          } /* fine del blocco if */
40          else { /* se il file temporaneo non può essere aperto */
41             printf("Unable to open temporary file\n");
42          } /* fine del blocco else */
43
44       } /* fine del blocco if */
45       else { /* se il file non può essere aperto */
46          printf("Unable to open %s\n", fileName);
47       } /* fine del blocco else */
48
49       return 0; /* indica che il programma è terminato con successo */
50
51    } /* fine della funzione main */
```

```
This program changes tabs to spaces.
Enter a file to be modified: data

The file before modification is:
0     1     2     3     4
      5     6     7     8     9

The file after modification is:
0 1 2 3 4
 5 6 7 8 9
```

Figura 14.6 File temporanei

14.10 La gestione dei segnali

Un evento inatteso, o *segnale*, può causare la chiusura prematura di un programma. Alcuni degli eventi inattesi sono: le *istruzioni illegali*, le *violazioni dei segmenti di memoria*, gli *interrupt* (premere `<ctrl> c` in un sistema UNIX o Windows), l'*ordine di chiusura da parte del sistema operativo* e le *eccezioni dei calcoli in virgola mobile* (divisione per zero o moltiplicazione di grandi valori in virgola mobile). La *libreria per la gestione dei segnali* (`signal.h`) fornisce, con la funzione *`signal`*, la possibilità di *intercettare* gli eventi inattesi. La funzione `signal` riceve due argomenti: un intero, che rappresenta il numero del segnale, e un puntatore alla funzione che lo gestirà. I segnali possono essere generati dalla funzione `raise`, che accetta come suo argomento un intero che rappresenta il numero del segnale. La Figura 14.7 riassume i segnali standard definiti nel file di intestazione *`signal.h`*. In Figura 14.8 viene mostrato l'uso delle funzioni `signal` e `raise`.

In Figura 14.8 verrà usata la funzione `signal` per intercettare un segnale interattivo (`SIGINT`). La riga 15 richiamerà `signal` con `SIGINT` e un puntatore a `signalHandler` (ricordate che il nome di una funzione è un puntatore all'inizio della stessa). Nel momento verrà ricevuto un segnale di tipo `SIGINT`, il controllo sarà passato alla funzione `signalHandler`, che visualizzerà un messaggio e darà all'utente la possibilità di scegliere se continuare la normale esecuzione del programma. Nel caso in cui l'utente scelga di continuare l'esecuzione, il

Segnale	Spiegazione
SIGABRT	Chiusura anormale del programma (per esempio, a causa di una chiamata alla funzione abort).
SIGFPE	Un'operazione aritmetica sbagliata, come una divisione per zero o un'operazione che provoca un overflow (eccedenza della capacità di memorizzazione dei dati).
SIGILL	Scoperta di un'istruzione illegale.
SIGINT	Ricezione di un segnale di attenzione interattivo.
SIGSEGV	Un accesso non valido alla memoria.
SIGTERM	Una richiesta di chiusura inviata al programma.

Figura 14.7 I segnali standard in signal.h

gestore del segnale sarà inizializzato nuovamente, richiamando ancora `signal` e il controllo ritornerà al punto del programma in cui era stato rilevato il segnale. In questo programma, sarà usata la funzione `raise` (riga 24) per simulare un segnale interattivo. Sarà scelto un numero casuale tra `1` e `50`. Nel caso in cui il numero sia `25`, allora sarà invocata `raise` per generare il segnale. Normalmente, i segnali interattivi sono inizializzati all'esterno del programma. Per esempio, premendo `<ctrl>` `c` durante l'esecuzione del programma in un sistema UNIX o Windows, sarà generato un segnale interattivo che terminerà l'esecuzione del programma. La gestione dei segnali può essere usata per intercettare i segnali interattivi ed evitare che il programma sia chiuso.

```
1  /* Fig. 14.8: fig14_08.c
2     Usare la gestione dei segnali */
3  #include <stdio.h>
4  #include <signal.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  void signalHandler(int signalValue); /* prototipo */
9
10 int main()
11 {
12    int i; /* contatore usato per eseguire un'iterazione 100 volte */
13    int x;  /* variabile per memorizzare valori casuali
                 compresi fra 1 e 50 */
14
15    signal(SIGINT, signalHandler); /* registra la funzione
                                        di gestione del segnale */
16    srand(clock());
17
18    /* manda in output I numeri da 1 a 100 */
19    for (i = 1; i <= 100; i++) {
20       x = 1 + rand() % 50; /* genera dei numeri casuali per
                                  segnalare SIGINT */
```

Figura 14.8 Gestione dei segnali (continua)

```
21
22          /* segnala SIGINT quando x vale 25 */
23          if (x == 25) {
24             raise(SIGINT);
25          } /* fine del comando if */
26
27          printf("%4d", i);
28
29          /* manda in output \n quando i è un multiplo di 10 */
30          if (i % 10 == 0) {
31             printf("\n");
32          } /* fine del comando if */
33
34       } /* fine del comando for */
35
36       return 0; /* indica che il programma è terminato con successo */
37
38    } /* fine della funzione main */
39
40    /* gestisce il segnale */
41    void signalHandler(int signalValue)
42    {
43       int response; /* risposta dell'utente al verificarsi
                          del segnale (1 o 2) */
44
45       printf("%s%d%s\n%s",
46              "\nInterrupt signal (", signalValue, ") received.",
47              "Do you wish to continue (1 = yes or 2 = no)? ");
48
49       scanf("%d", &response);
50
51       /* controllo sulla validità delle risposte */
52       while (response != 1 && response != 2) {
53          printf("(1 = yes or 2 = no)? ");
54          scanf("%d", &response);
55       } /* fine del comando while */
56
57       /* determina se è il momento di terminare l'esecuzione */
58       if (response == 1) {
59
60          /* registra il gestore del segnale per il prossimo SIGINT */
61          signal(SIGINT, signal_handler);
62       } /* fine del blocco if */
63       else {
64          exit(EXIT_SUCCESS);
65       } /* fine del blocco else */
66
67    } /* fine della funzione signalHandler */
```

Figura 14.8 Gestione dei segnali (continua)

```
  1   2   3   4   5   6   7   8   9  10
 11  12  13  14  15  16  17  18  19  20
 21  22  23  24  25  26  27  28  29  30
 31  32  33  34  35  36  37  38  39  40
 41  42  43  44  45  46  47  48  49  50
 51  52  53  54  55  56  57  58  59  60
 61  62  63  64  65  66  67  68  69  70
 71  72  73  74  75  76  77  78  79  80
 81  82  83  84  85  86  87  88  89  90
 91  92  93
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
 94  95  96

Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2
```

Figura 14.8 Gestione dei segnali

14.11 Allocazione dinamica della memoria: le funzioni calloc e realloc

Il Capitolo 12, "Le strutture di dati", ha introdotto la nozione di allocazione dinamica della memoria mediante l'utilizzo della funzione `malloc`. Come abbiamo affermato nel Capitolo 12, i vettori sono migliori delle liste concatenate per rapidità di ordinamento, ricerca e accesso ai dati. Tuttavia, i vettori sono normalmente delle *strutture di dati statiche*. La libreria di utilità generiche (`stdlib.h`) fornisce altre due funzioni per l'allocazione dinamica della memoria: `calloc` e `realloc`. Queste possono essere usate per creare e modificare dei *vettori dinamici*. Come mostrato nel Capitolo 7, I puntatori del C, un puntatore a un vettore può essere indicizzato proprio come quest'ultimo. Di conseguenza, un puntatore a una porzione contigua di memoria creata con la funzione `calloc` può essere gestito come un vettore. La funzione `calloc` alloca dinamicamente la memoria di un vettore. Il suo prototipo è

```
void *calloc(size_t nmemb, size_t size);
```

I suoi due argomenti rappresentano il numero degli elementi (`nmemb`) e la loro dimensione (`size`). Inoltre la funzione `calloc` inizializza a zero gli elementi del vettore. La funzione restituisce un puntatore alla memoria allocata, o un `NULL` qualora questa non sia stata allocata. La differenza principale tra malloc e calloc consiste nel fatto che `calloc` azzera la memoria che alloca, mentre `malloc` non lo fa.

La funzione `realloc` cambia la dimensione di un oggetto allocato da una precedente chiamata a `malloc`, `calloc` o `realloc`. I contenuti originali dell'oggetto non saranno modificati, purché la quantità di memoria allocata sia maggiore di quella allocata in precedenza. In caso contrario, solo i contenuti compresi nella dimensione del nuovo oggetto non saranno cambiati. Il prototipo per la funzione `realloc` è

```
void *realloc(void *ptr, size_t size);
```

I due argomenti sono un puntatore all'oggetto originale (`ptr`) e la nuova dimensione dello stesso (`size`). Nel caso in cui `ptr` sia `NULL`, `realloc` funzionerà allo stesso modo di `malloc`. Nel caso in cui `size` sia `0` e `ptr` sia `NULL`, la memoria allocata per l'oggetto sarà rilasciata. Altrimenti, qualora `ptr` non sia `NULL` e la dimensione sia maggiore di zero, `realloc` tenterà di allocare per l'oggetto un nuovo blocco di memoria. Nel caso in cui il nuovo spazio non possa essere allocato, l'oggetto puntato da `ptr` non sarà modificato. La funzione `realloc` restituisce un puntatore alla memoria riallocata o un `NULL` per indicare che la memoria non è stata riallocata.

14.12 Il salto incondizionato con goto

Nel corso di questo testo, abbiamo posto l'accento sull'importanza di usare le tecniche di programmazione strutturata per costruire un software affidabile, per il quale le attività di messa a punto, manutenzione e modifica siano facili. In alcune occasioni, le prestazioni potranno essere più importanti della stretta aderenza alle tecniche di programmazione strutturata. Nelle suddette occasioni, potranno essere utilizzate alcune tecniche di programmazione non strutturata. Per esempio, potremo utilizzare `break` per terminare l'esecuzione di una struttura di iterazione, prima che la condizione di continuazione diventi falsa. Ciò eviterà delle inutili ripetizioni del ciclo, nel caso in cui l'attività sia già stata completata prima del termine del ciclo.

Un altro esempio di programmazione non strutturata è l'*istruzione `goto`*: un salto incondizionato. Il risultato di un'istruzione `goto` sarà la deviazione del flusso di controllo del programma alla prima istruzione successiva alla *label* (*etichetta*) specificata nell'istruzione `goto`. Un'etichetta è un identificatore seguito dal carattere due punti. Essa deve comparire nella stessa funzione dell'istruzione `goto` che vi fa riferimento. In Figura 14.9 si userà l'istruzione `goto` per iterare dieci volte e, in ognuna, visualizzare il valore del contatore. Dopo averlo inizializzato a `1`, la riga 11 controllerà `count` per determinare se questo sia maggiore di `10` (l'etichetta `start` sarà saltata, perché le etichette non eseguono alcuna azione). In questo caso, il controllo sarà trasferito dalla `goto` alla prima istruzione successiva all'etichetta `end` (che compare in riga 20). In caso contrario, le righe 15-16 visualizzeranno e incrementeranno `count` e il controllo sarà trasferito dalla `goto` (riga 18) alla prima istruzione successiva all'etichetta `start` (che compare in riga 9).

```
1  /* Fig. 14.9: fig14_09.c
2     Usare goto */
3  #include <stdio.h>
4
5  int main()
6  {
7     int count = 1; /* inizializza count */
8
9     start:                    /* etichetta */
10
11       if (count > 10) {
12          goto end;
13       } /* fine del comando if */
```

Figura 14.9 Il comando goto (continua)

```
14
15          printf("%d  ", count);
16          ++count;
17
18          goto start; /* salta all'etichetta start in riga 9 */
19
20       end:                        /* etichetta */
21          putchar('\n');
22
23       return 0; /* indica che il programma è terminato con
                      successo */
24
25    } /* fine della funzione main */
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 14.9 Il comando goto

Nel Capitolo 3, abbiamo affermato che per scrivere qualsiasi programma sono necessarie soltanto tre strutture di controllo: la sequenza, la selezione e l'iterazione. Attenendosi alle regole della programmazione strutturata, sarà possibile creare strutture di controllo profondamente nidificate, dalle quali però sarà difficile uscire in modo efficiente. In situazioni di questo genere, alcuni programmatori usano l'istruzione `goto` come un'uscita rapida da una struttura profondamente nidificata. Ciò eliminerà la necessità di verificare le molteplici condizioni di uscita da una struttura di controllo.

Obiettivo efficienza 14.3

L'istruzione `goto` *può essere usata per uscire in modo efficiente da strutture di controllo profondamente nidificate.*

Ingegneria del software 14.3

L'istruzione `goto` *dovrebbe essere usata soltanto in applicazioni orientate alle prestazioni. L'istruzione* `goto` *non è strutturata e può condurre a programmi in cui la messa a punto, la manutenzione e le modifiche siano più difficili.*

Esercizi di autovalutazione

14.1 Riempite gli spazi vuoti in ognuna delle seguenti righe:

a) Il simbolo _____ è usato per redirezionare l'input dei dati in modo che provenga da un file piuttosto che dalla tastiera.

b) Il simbolo _____ è usato per redirezionare l'output dello schermo in modo che sia memorizzato in in un file.

c) Il simbolo _____ è usato per accodare alla fine di un file l'output di un programma.

d) Un _____ è usato per redirezionare l'output di un programma sull'input di un altro.

e) Un'_____, nell'elenco degli argomenti di una funzione, indica che questa può ricevere un numero variabile di argomenti.

f) La macro _____ deve essere invocata, prima di accedere agli elementi di un elenco variabile di argomenti.
g) La macro _____ è usata per accedere ai singoli elementi di un elenco variabile di argomenti.
h) La macro _____ facilita un normale ritorno da una funzione il cui elenco variabile di argomenti è stato puntato dalla macro `va_start`.
i) L'argomento _____ della funzione `main` riceve il numero di quelli presenti sulla riga di comando.
j) L'argomento _____ della funzione `main` immagazzina quelli della riga di comando in stringhe di caratteri.
k) Il programma di utilità UNIX _____ legge un file chiamato _____, che contiene le istruzioni per la compilazione e il linking di un programma suddiviso in molti file sorgente. Il programma di utilità compila nuovamente un file, solo qualora questo sia stato modificato dopo la sua ultima compilazione.
l) La funzione _____ forza un programma a terminare la propria esecuzione.
m) La _____ registra una funzione che sarà invocata dopo la normale chiusura del programma.
n) Un _____ può essere accodato a una costante intera o in virgola mobile per specificare il tipo esatto della stessa.
o) La funzione _____ apre un file temporaneo che esisterà fino alla sua chiusura o fino al termine dell'esecuzione del programma.
p) La funzione _____ può essere usata per intercettare un evento inatteso.
q) La funzione _____ genera un segnale all'interno di un programma.
r) La funzione _____ alloca dinamicamente la memoria di un vettore e inizializza a zero gli elementi.
s) La funzione _____ modifica la dimensione di un blocco di memoria precedentemente allocato in modo dinamico.

Risposte agli esercizi di autovalutazione

14.1 a) redirezionamento dell'input (`<`). b) redirezionamento dell'output (`>`). c) accodamento dell'output (`>>`). d) pipe (`|`). e) ellissi (`...`). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make`, `makefile`. l) `exit`. m) `atexit`. n) suffisso. o) `tmpfile`. p) `signal`. q) `raise`. r) `calloc`. s) `realloc`.

Esercizi

14.2 Scrivete un programma che calcoli il prodotto di una serie di interi passati alla funzione `product`, usando un elenco variabile di argomenti. Verificate la vostra funzione con diverse chiamate, ognuna con un numero differente di argomenti.

14.3 Scrivete un programma che visualizzi gli argomenti immessi sulla riga di comando di un programma.

14.4 Scrivete un programma che ordini un vettore di interi in modo ascendente o discendente. Il programma dovrà usare gli argomenti della riga di comando per passare `-a`, per un ordinamento ascendente, o `-d` per uno discendente. [*Nota*: in UNIX questo è il formato standard per il passaggio delle opzioni a un programma].

14.5 Scrivete un programma che inserisca uno spazio tra ogni carattere di un file. Il programma dovrà prima scrivere il contenuto del file da modificare in un archivio temporaneo, inserendo gli spazi tra i caratteri, e quindi dovrà ricopiare il file temporaneo su quello originale. Questa operazione dovrà sostituire i contenuti del file originale.

14.6 Consultate il manuale del vostro sistema per determinare quali segnali siano supportati dalla relativa libreria di gestione (`signal.h`). Scrivete un programma che contenga alcune gestioni dei segnali standard `SIGABRT` e `SIGINT`. Il programma dovrà controllare l'intercettazione di questi segnali richiamando la funzione `abort`, per generarne uno di tipo `SIGABRT`, e premendo `<ctrl> c`, per generarne uno di tipo `SIGINT`.

14.7 Scrivete un programma che allochi dinamicamente un vettore di interi. La dimensione del vettore dovrà essere immessa dalla tastiera. Gli elementi del vettore dovranno essere impostati con valori immessi dalla tastiera. Visualizzate i valori del vettore e quindi allocate nuovamente la sua memoria a metà del numero corrente di elementi. Visualizzate i valori rimasti nel vettore per confermare che corrispondano alla prima metà dei valori contenuti nel vettore originale.

14.8 Scrivete un programma che riceva come argomenti due nomi di file dalla riga di comando, legga uno per volta i caratteri dal primo e li scriva in ordine inverso nel secondo.

14.9 Scrivete un programma che usi l'istruzione `goto` per simulare una struttura di iterazione nidificata che visualizzi un quadrato di asterischi come il seguente:

```
*****
*   *
*   *
*   *
*****
```

Il programma dovrà usare soltanto le seguenti tre istruzioni `printf`:

```
printf("*");
printf(" ");
printf("\n");
```

Capitolo 15

Introduzione al C99

Obiettivi

- Conoscere molte delle nuove caratteristiche dello standard C99.
- Conoscere alcune caratteristiche chiave del C99 inserite nel contesto di programmi completi e funzionanti.
- Essere in grado di utilizzare i commenti introdotti da `//`.
- Essere in grado di intercalare dichiarazioni e codice eseguibile e di dichiarare variabili nelle intestazioni dei comandi `for`.
- Essere in grado di inizializzare vettori e strutture con gli inizializzatori designati.
- Essere in grado di utilizzare il tipo di dati `bool` per creare variabili booleane i cui valori possono essere `true` o `false`.
- Essere in grado di creare vettori di lunghezza variabile e di passarli come argomenti alle funzioni.
- Essere in grado di compiere operazioni aritmetiche su variabili complesse.

15.1 Introduzione

Il *C99* è uno standard emendato per il linguaggio di programmazione C che raffina ed espande le potenzialità dello Standard C. Tuttavia il C99 non è stato ancora largamente adottato e non tutti i principali compilatori lo supportano; tra questi, molti supportano solo un sottoinsieme del linguaggio. Per tali ragioni abbiamo deciso di trattare il C99 soltanto in quest'ultimo capitolo "opzionale", mentre i Capitoli 1-14 parlano del C89 (la versione iniziale dello standard ANSI/ISO), che è quasi universalmente adottato.

In questo capitolo proponiamo un'introduzione al C99, discutendo il supporto offerto dal compilatore ed includendo i collegamenti web a diversi compilatori ed ambienti integrati (IDE) gratuiti che supportano il C99 a vari livelli. Illustreremo, mediante esempi di codice completo e funzionante, alcune delle caratteristiche chiave del C99, compresi i commenti introdotti da `//`, la combinazione intercalata di dichiarazioni e codice eseguibile, le dichiarazioni nei comandi `for`, gli inizializzatori designati, i letterali composti, il tipo di dati `bool`, il tipo di ritorno implicito `int` nei prototipi di funzione e nelle definizioni di funzione (non permesso nel C99), i numeri complessi ed i vettori di lunghezza variabile. Forniremo inoltre brevi spiegazioni relative ad alcune caratteristiche chiave aggiuntive del C99, includendo gli identificatori estesi, i puntatori ristretti, la divisione intera affidabile, i vettori flessibili come membri delle strutture, il tipo di dati `long long int`, il supporto della matematica indipendente dal tipo, le funzioni inline, il comando di ritorno senza espressione e la funzione `snprintf`.

Un'ulteriore caratteristica significativa è l'aggiunta di versioni `float` e `long double` della maggior parte delle funzioni matematiche dichiarate in `<math.h>`. Includeremo infine un esaustivo elenco di risorse internet e web per facilitare il reperimento di compilatori ed ambienti integrati (IDE) con supporto del C99 e per aiutare a "scavare" in profondità nei dettagli tecnici del linguaggio.

15.2 Supporto del C99

La maggior parte dei compilatori C e C++ non supportavano il C99 al momento del suo rilascio. In seguito il supporto è cresciuto e molti compilatori sono ormai vicini ad essere conformi allo standard C99. Microsoft Visual C++ 2005 Express Edition non supporta il C99; per avere maggiori informazioni sulla futura conformità allo standard C99 di Microsoft Visual C++, si consiglia di visitare l'indirizzo web `msdn.microsoft.com/chats/transcripts/vstudio/vstudio_022703.aspx`.

Nel corso di questo capitolo faremo uso dell'IDE Dev-C++ 4.9.9.2 della Bloodshed Software (`www.bloodshed.net/dev/devcpp.html`). Questo ambiente integrato utilizza come compilatore predefinito il MingW (Minimalist GNU for Windows), una versione del compilatore GNU GCC (`gcc.gnu.org/install`), che supporta la maggior parte del C99. Per specificare di adottare lo standard C99 durante la compilazione, è necessario editare l'area di testo denominata *Add the following commands when calling compiler:*, aggiungendo la stringa `-std=c99`, nella finestra *Compiler Options* (Figura 15.1) richiamabile dal menu *Tools*. Bisogna inoltre assicurarsi che sia presente il segno di spunta a sinistra dell'etichetta dell'area di testo.

Per eseguire i programmi scritti in Dev-C++, bisogna aprire un prompt dei comandi di Windows, utilizzare il comando `cd` (change directory) per posizionarsi nella directory in cui il programma è stato compilato e digitare il nome del programma per eseguirlo (in questo caso `fig15_03.exe`).

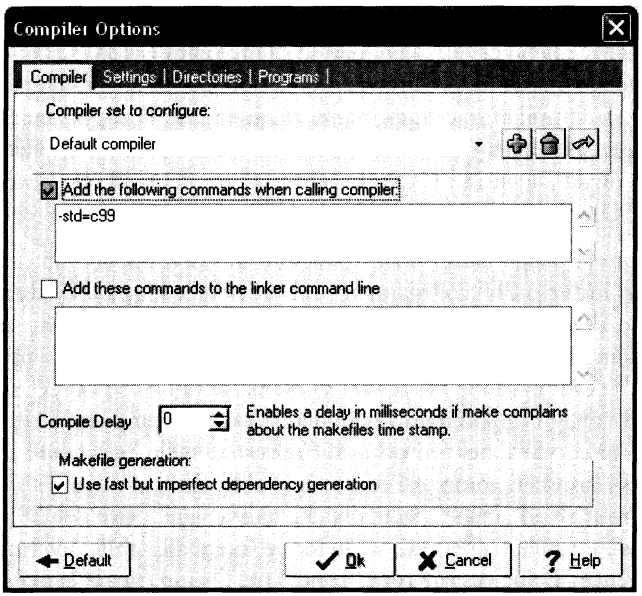


Figura 15.1 La finestra Compiler Options in Dev-C++ 4.9.9.2

Il compilatore Comeau C/C++ è pienamente conforme allo standard C99 e viene messo a disposizione dalla Comeau Computing che lo vende sul suo sito web: `www.comeaucomputing.com`. Il sito, inoltre, mette a disposizione la documentazione ed una versione online gratuita del compilatore che serve a mostrare quali siano i messaggi d'errore e di avvertimento generati dal Comeau C/C++ (questa versione tuttavia non permette di eseguire i programmi). Il compilatore Comeau C/C++ può essere utilizzato congiuntamente alla libreria standard Dinkum C99 della Dinkumware (`www.dinkumware.com/c99.aspx`). Questa libreria supporta tutti i nuovi file header richiesti dal C99 e dal C95 e può essere utilizzata anche con altri compilatori, incluso il Microsoft Visual C++.

15.3 I nuovi file header del C99

In Figura 15.2 sono elencati in ordine alfabetico i file header della libreria standard aggiunti nel C99 (tre di questi erano stati aggiunti nel C95).

File header della libreria standard	Spiegazione
`<complex.h>`	Contiene macro e prototipi di funzione per il supporto dei numeri complessi (si veda la Sezione 15.10). [Caratteristica del C99.]
`<fenv.h>`	Fornisce delle informazioni sulle potenzialità e sull'ambiente del calcolo a virgola mobile dell'implementazione in C. [Caratteristica del C99.]
`<inttypes.h>`	Definisce diversi nuovi tipi interi portabili e mette a disposizione degli specificatori di formato per i tipi definiti. [Caratteristica del C99.]
`<iso646.h>`	Definisce delle macro che rappresentano gli operatori di uguaglianza, di relazione e bit a bit; un'alternativa alla rappresentazione tramite "trigrafi". [Caratteristica del C95.]
`<stdbool.h>`	Contiene le macro che definiscono `bool`, `true` e `false`, utilizzate per le variabili booleane (si veda la Sezione 15.8). [Caratteristica del C99.]
`<stdint.h>`	Definisce i tipi interi estesi e le macro ad essi relative. [Caratteristica del C99.]
`<tgmath.h>`	Fornisce delle macro indipendenti dal tipo che permettono di applicare le funzioni del file header `<math.h>` a diversi tipi di parametri (si veda la Sezione 15.12). [Caratteristica del C99.]
`<wchar.h>`	Congiuntamente al file header `<wctype.h>`, fornisce il supporto per l'input e l'output con caratteri multibyte e wide. [Caratteristica del C95.]
`<wctype.h>`	Congiuntamente al file `<wchar.h>`, fornisce il supporto per la libreria sui caratteri wide. [Caratteristica del C95.]

Figura 15.2 File header della libreria standard aggiunti nel C99 e nel C95

15.4 Commenti introdotti da //

Il C99 permette di introdurre i commenti con // (come avviene in C++, Java e C#). Ogni volta che la sequenza di caratteri // occorre all'esterno delle virgolette, il resto della riga viene considerato un commento (si veda la Figura 15.3; righe 1, 2, 7, 10, 13 e 14).

15.5 Intercalare dichiarazioni e codice eseguibile

Il C89 richiede che tutte le variabili con visibilità nel blocco siano dichiarate all'inizio di quest'ultimo. Il C99 *consente di intercalare dichiarazioni e codice eseguibile*. Una variabile può essere dichiarata ovunque in un blocco prima del suo utilizzo. Si analizzi ad esempio il programma C99 riportato in Figura 15.3.

```
1 // Fig 15.3: fig15_03.c
2 // Intercalare dichiarazioni e codice eseguibile in C99
3 #include <stdio.h>
4
5 int main( void )
6 {
7   int x = 1 ; // dichiarazione di una variabile all'inizio di un blocco
8   printf( "x is %d\n", x );
9
10  int y = 2 ; // dichiara una variabile in mezzo al codice eseguibile
11  printf( "y is %d\n", y );
12
13  return 0 ; // indica che il programma è terminato con successo
14 } // fine della funzione main
```

```
x is 1
y is 2
```

Figura 15.3 Intercalare dichiarazioni e codice eseguibile in C99.

In questo programma la funzione `printf` (codice eseguibile) viene richiamata in riga 8, mentre in riga 10 viene dichiarata la variabile `y` di tipo `int`: ciò non è permesso in C89. In C99 invece è possibile dichiarare le variabili vicino al punto del loro primo utilizzo, anche nel caso in cui tali dichiarazioni compaiano in seguito a del codice eseguibile all'interno di un blocco. Non viene dichiarata `int y` (riga 10) fino ad un attimo prima del suo utilizzo (riga 11). Nonostante ciò migliori la leggibilità del programma e riduca la possibilità di riferimenti non voluti, molti programmatori continuano a preferire il raggruppamento delle loro dichiarazioni di variabili all'inizio dei blocchi.

Una variabile non può essere dichiarata dopo il codice che la utilizza. In Figura 15.4 viene dichiarata `int y` (riga 12) dopo un comando eseguibile che tenta di visualizzare a schermo `y` (riga 10). Ciò dà luogo ad un errore di compilazione.

```
1 // Fig 15.4: fig15_04.c
2 // Tentativo di dichiarare una variabile dopo il suo utilizzo in C99
3 #include <stdio.h>
4
5 int main( void )
6 {
7    int x = 1 ; // dichiarazione di x
8
9    printf( "The value of x is %d\n", x ); // visualizza x
10   printf( "The value of y is %d\n", y ); // visualizza y (errore)
11
12   int y = 2 ; // dichiarazione di y
13
14   return 0 ; // indica che il programma è terminato con successo
15 } // fine della funzione main
```

```
../examples/ch15/fig15_03.c: In function 'main':
../examples/ch15/fig15_03.c:10: error: 'y' undeclared
(first use in this function)
../examples/ch15/fig15_03.c:10: error:
(Each undeclared identifier is reported only once
../examples/ch15/fig15_03.c:10: error: for each function it appears in.)
```

Figura 15.4 Tentativo di dichiarazione di una variabile dopo il suo utilizzo in C99. [Errore prodotto dal compilatore MingW utilizzando l'IDE Dev C++.]

15.6 Dichiarare una variabile nell'intestazione di un comando for

Come ricorderete, un comando `for` consiste in un'inizializzazione, una condizione di continuazione del ciclo ed un corpo del ciclo. In Figura 15.5 è riportato un programma C89 che utilizza un comando `for` per visualizzare i numeri da 1 a 5.

Il C99 estende la definizione del comando `for` in C89, permettendo alla clausola di inizializzazione di includere una dichiarazione. Invece di utilizzare una variabile esistente come contatore del ciclo, si può creare una nuova variabile come contatore del ciclo nell'intestazione del comando `for`, limitandone la visibilità all'interno di quest'ultimo. Il programma C99 riportato in Figura 15.6 dichiara una variabile nell'intestazione del comando `for`.

Qualsiasi variabile dichiarata in un comando `for` ha la visibilità limitata al comando `for`, ovvero, la variabile non esiste al di fuori del comando `for`. In Figura 15.7 viene riportato un tentativo fallito di accedere ad una variabile di questo tipo dopo la fine del corpo del comando.

```
1 // Fig 15.5: fig15_05.c
2 // Dichiarare un contatore del ciclo prima di un comando for in C89
3 #include <stdio.h>
4
```

Figura 15.5 Dichiarare un contatore del ciclo prima di un comando for in C89 (continua)

```
 5 int main( void )
 6 {
 7    int x; // dichiara il contatore del ciclo
 8
 9    // visualizza i valori da 1 a 5
10    printf( "Values of x\n" );
11    for ( x = 1 ; x <= 5 ; x++ ) // inizializza il contatore del ciclo
12    printf( "%d\n", x );
13
14    printf( "Value of x is %d\n", x ); // x è ancora visibile
15
16    return 0 ; // indica che il programma è terminato con successo
17 } // fine della funzione main
```

```
Values of x
1
2
3
4
5
Value of x is 6
```

Figura 15.5 Dichiarare un contatore del ciclo prima di un comando for in C89

```
 1 // Fig 15.6: fig15_06.c
 2 // Dichiarare una variabile nell'intestazione di un comando for in C99
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    printf( "Values of x\n" );
 8
 9    // dichiara una variabile nell'intestazione di un comando for
10    for ( int x = 1 ; x <= 5 ; x++ )
11          printf( "%d\n", x );
12
13    return 0 ; // indica che il programma è terminato con successo
14 } // fine della funzione main
```

```
Values of x
1
2
3
4
5
```

Figura 15.6 Dichiarare una variabile nell'intestazione di un comando for in C99

```
 1 // Fig 15.7: fig15_05.c
 2 // Accedere a una variabile di un comando for al di fuori
   // del comando for in C99
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    printf( "Values of x:\n" );
 8
 9    // dichiara una variabile nell'intestazione del comando for
10    for ( int x = 1 ; x <= 5 ; x++ )
11          printf( "%d\n", x );
12
13    printf( "Value of x is: %d\n", x ); // x non è più visibile
14
15    return 0 ; // indica che il programma è terminato con successo
16 } // fine della funzione main
```

```
../examples/ch15/fig15_07.c: In function 'main':
../examples/ch15/fig15_07.c:13: error: 'x' undeclared
(first use in this function)
../examples/ch15/fig15_07.c:13: error:
(Each undeclared identifier is reported only once
../examples/ch15/fig15_07.c:13: error: for each function it appears in.)
```

Figura 15.7 Accedere ad una variabile di un comando for al di fuori del comando for in C99.
[Errore prodotto dal compilatore MingW utilizzando l'IDE Dev-C++.]

15.7 Inizializzatori designati e letterali composti

Gli inizializzatori designati permettono di inizializzare gli elementi di un vettore, una union od una struct facendovi esplicito riferimento tramite l'indice od il nome. In Figura 15.8 è riportato come sia possibile assegnare il primo e l'ultimo elemento di un vettore in C89.

```
 1 // Fig 15.8: fig15_08.c
 2 // Assegnare dei valori agli elementi di un vettore in C89
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int i; // dichiara un contatore del ciclo
 8    int a[ 5 ]; // dichiara un vettore
 9
10    a[ 0 ] = 1 ; // assegna esplicitamente dei valori agli elementi
                   // del vettore...
11    a[ 4 ] = 2 ; // dopo la dichiarazione del vettore stesso
12
```

Figura 15.8 Assegnare dei valori agli elementi di un vettore in C89 (continua)

```
13    // azzera tutti gli elementi tranne il primo e l'ultimo
14    for ( i = 1 ; i < 4 ; i++ )
15          a[ i ] = 0 ;
16
17    // visualizza il contenuto del vettore
18    printf( "The array is \n" );
19    for ( i = 0; i < 5 ; i++ )
20          printf( "%d\n", a[ i ] );
21
22    return 0 ; // indica che il programma è terminato con successo
23 } // fine della funzione main
```

```
The array is
1
0
0
0
2
```

Figura 15.8 Assegnare dei valori agli elementi di un vettore in C89

In Figura 15.9 viene nuovamente riportato il programma, ma, invece di assegnare dei valori al primo ed all'ultimo elemento del vettore, essi vengono inizializzati esplicitamente tramite gli indici, utilizzando degli inizializzatori designati.

```
 1 // Fig 15.9: fig15_09.c
 2 // Utilizzare degli inizializzatori designati
 3 // per inizializzare gli elementi di un vettore in C99
 4 #include <stdio.h>
 5
 6 int main( void )
 7 {
 8    int a[5 ] =
 9    {
10          [ 0 ] = 1 , // inizializza gli elementi per mezzo
                        // degli inizializzatori designati...
11          [ 4 ] = 2 // nella dichiarazione del vettore
12    }; // il carattere punto e virgola è necessario
13
14     // visualizza il contenuto del vettore
15    printf( "The array is \n" );
16    for ( int i = 0 ; i < 5 ; i++ )
17          printf( "%d\n", a[ i ] );
18
19    return 0 ; // indica che il programma è terminato con successo
20 } // fine della funzione main
```

Figura 15.9 Utilizzare degli inizializzatori designati per inizializzare gli elementi di un vettore in C99 (continua)

```
The array is
1
0
0
0
2
```

Figura 15.9 Utilizzare degli inizializzatori designati per inizializzare gli elementi di un vettore in C99

Le righe 8-12 dichiarano un vettore ed inizializzano gli elementi specificati fra parentesi graffe. Si noti la sintassi: ogni inizializzatore nella lista di inizializzatori (righe 10-11) è separato dal successivo da una virgola e la parentesi graffa di chiusura è seguita da un punto e virgola. Gli elementi che non sono esplicitamente inizializzati sono implicitamente impostati a zero (del tipo corretto). Questa sintassi non è consentita in C89.

Una lista di inizializzatori può essere utilizzata, oltre che per dichiarare una variabile, anche per creare un vettore, una `struct` od una `union` senza nome. Questo fatto è noto come letterale composto. Per esempio, se si vuole passare alla funzione `demoFunction` un vettore equivalente al vettore `a` riportato in Figura 15.9, ma senza dover dichiarare il vettore `a` nel programma, si può utilizzare la seguente sintassi:

```
demoFunction( ( int [ 5 ] ) { [ 0 ] = 1, [ 4 ] = 2 } );
```

Si consideri l'esempio maggiormente elaborato riportato in Figura 15.10, dove si utilizza un inizializzatore designato per un vettore di `struct`.

In riga 17 viene utilizzato un inizializzatore designato per inizializzare esplicitamente un elemento `struct` del vettore. In seguito, all'interno di questa inizializzazione, viene usato un ulteriore livello di inizializzatori designati per inizializzare esplicitamente i membri `x` e `y` della `struct`. Per inizializzare i membri di una `struct` o di una `union` si elenca il nome di ogni membro preceduto da un punto.

```
 1 // Figura 15.10: fig15_10.c
 2 // Utilizzare gli inizializzatori designati per inizializzare
   // un vettore di struct in C99
 3 #include <stdio.h>
 4
 5 struct twoInt // dichiara una struct composta da due interi
 6 {
 7    int x;
 8    int y;
 9 }; // fine della struct twoInt
10
11 int main( void )
12 {
13    // inizializza esplicitamente gli elementi del vettore a quindi
14    // inizializza esplicitamente i membri di ogni elemento della struct
```

Figura 15.10 Utilizzare gli inizializzatori designati per inizializzare un vettore di struct in C99 (continua)

```
15    struct twoInt a[ 5 ] =
16    {
17          [ 0 ] = { .x = 1 , .y = 2 },
18          [ 4 ] = { .x = 10 , .y = 20 }
19    };
20
21    // visualizza il contenuto del vettore
22    printf( "\nx\ty\n" );
23    for ( int i = 0 ; i <= 4 ; i++ )
24          printf( "%d\t%d\n", a[ i ].x, a[ i ].y );
25
26    return 0 ; // indica che il programma è terminato con successo
27 } // fine della funzione main
```

```
x     y
1     2
0     0
0     0
0     0
10    20
```

Figura 15.10 Utilizzare gli inizializzatori designati per inizializzare un vettore di struct in C99

Si confrontino le righe 15-19 riportate in Figura 15.10, che utilizzano degli inizializzatori designati, al codice eseguibile seguente, che non fa uso degli inizializzatori designati:

```
struct twoInt a[ 5 ];

a[ 0 ].x = 1;
a[ 0 ].x = 2;
a[ 4 ].x = 10;
a[ 4 ].x = 20;
```

15.8 Il tipo di dati bool

Il *tipo di dati booleano* del C99 è `_Bool`, che può rappresentare soltanto i valori `0` oppure `1`. Si ricordi la convenzione del C di utilizzare lo zero ed i valori diversi da zero per rappresentare il falso ed il vero: il valore `0` in una condizione corrisponde al falso, mentre ogni valore diverso da zero in una condizione corrisponde al vero. Assegnare un valore diverso da zero ad una variabile di tipo `_Bool` la imposta a `1`. Il C99 fornisce il file header `<stdbool.h>` che definisce delle macro per rappresentare il tipo di dati `bool` ed i suoi valori (`true` e `false`). Queste macro rimpiazzano `true` con `1`, `false` con `0` e `bool` con la parola chiave del C99 `_Bool`. In Figura 15.11 viene utilizzata una funzione chiamata `isEven` (righe 33-41) che restituisce un valore di tipo `bool` per indicare se il numero fornito in input dall'utente sia pari o dispari.

In riga 11 viene dichiarata una variabile di tipo `bool` chiamata `valueIsEven`; le righe 15-16 all'interno del ciclo chiedono all'utente di inserire un numero intero e leggono tale numero. In riga 18 il valore in input viene passato alla funzione `isEven` (righe 33-41). Si noti

che `isEven` restituisce un valore di tipo `bool`. In riga 35 viene utilizzato l'operatore di resto per determinare se l'argomento sia divisibile per 2. In caso positivo in riga 36 viene restituito il valore `true` (cioè, il numero è pari); altrimenti, in riga 39 viene restituito il valore `false` (cioè, il numero è dispari). Il risultato viene assegnato alla variabile `valueIsEven` di tipo `bool` in riga 18. In riga 21 tale variabile viene utilizzata come condizione in un comando `if...else`. Se `valueIsEven` vale `true`, in riga 22 viene visualizzata una stringa indicante che il valore è pari. Se `valueIsEven` vale `false`, in riga 25 viene visualizzata una stringa indicante che il valore è dispari.

```
1 // Fig 15.11: fig15_11.c
2 // Utilizzare il tipo di dati booleano ed i valori true e false in C99.
3 #include <stdio.h>
4 #include <stdbool.h> // consente l'utilizzo di bool, true e false
5
6 bool isEven( int number ); // prototipo di funzione
7
8 int main( void )
9 {
10    int input; // valore inserito dall'utente
11    bool valueIsEven; // memorizza il risultato della funzione isEven
12
13    // cicla per acquisire 2 input
14    for ( int i = 0 ; i < 2 ; i++ ) {
15          printf( "Enter an integer: " );
16          scanf( "%d", &input );
17
18          valueIsEven = isEven( input ); // determina se l'input è pari
19
20          // determina se l'input è pari
21          if ( valueIsEven ) {
22                printf( "%d is even \n\n", input );
23          } // fine del blocco if
24          else {
25                printf( "%d is odd \n\n", input );
26          } // fine del blocco else
27    } // fine del comando for
28
29    return 0 ;
30 } // fine della funzione main
31
32 // even restituisce true se number è pari
33 bool isEven( int number )
34 {
35    if ( number % 2 == 0 ) { // number è divisibile per 2?
36          return true ;
37    }
38    else {
39          return false ;
```

Figura 15.11 Utilizzare il tipo di dati bool ed i valori true e false in C99 (continua)

```
40    }
41 } // fine della funzione isEven
```

```
Enter an integer: 34
34 is even
Enter an integer: 23
23 is odd
```

Figura 15.11 Utilizzare il tipo di dati bool ed i valori true e false in C99

15.9 Tipo int implicito nelle dichiarazioni di funzioni

In C89 se una funzione non ha un tipo di ritorno esplicito, essa restituisce implicitamente un valore di tipo `int`. Inoltre, se una funzione non ha il tipo di un parametro specificato, questo viene dichiarato come `int`. Si consideri il programma riportato in Figura 15.12.

Nel momento in cui tale programma viene eseguito nell'ambiente Microsoft Visual C++ 2005 Express Edition, che non è conforme allo standard C99, non vengono generati errori o messaggi di avvertimento ed il programma viene eseguito correttamente. Il C99 non permette l'utilizzo implicito del tipo di dati `int`, richiedendo che i compilatori conformi allo standard C99 producano un avvertimento od un errore. Nel momento in cui lo stesso programma viene eseguito nell'IDE Dev-C++ utilizzando il compilatore MingW, che è conforme allo standard C99 per la maggior parte, si ottengono i messaggi d'avvertimento riportati in Figura 15.13.

Nonostante il programma venga eseguito, è importante che vengano notati i messaggi di avvertimento ed il programmatore si astenga dall'utilizzare delle funzioni con tipi `int` impliciti. Alcuni compilatori considerano la presenza di funzioni con tipi `int` impliciti degli errori di compilazione, impedendo che il programma venga eseguito.

```
 1 // Fig 15.12: fig15_12.c
 2 // Utilizzare implicitamente il tipo int in C89
 3 #include <stdio.h>
 4
 5 returnImplicitInt(); // prototipo con tipo di ritorno non specificato
 6 int demoImplicitInt(x); // prototipo con tipo del parametro
                           // non specificato
 7
 8 int main( void )
 9 {
10    int x;
11    int y;
12
13    // assegna dei dati con tipo di ritorno non specificato
      // a una variabile di tipo int
```

Figura 15.12 Utilizzare implicitamente il tipo int in C89 (continua)

```
14    x = returnImplicitInt();
15
16    // passa un int ad una funzione con tipo non specificato
17    y = demoImplicitInt( 82 );
18
19
20    printf( "x is %d\n", x );
21    printf( "y is %d\n", y );
22    return 0 ; // indica che il programma è terminato con successo
23
24 } // fine della funzione main
25
26 returnImplicitInt()
27 {
28    return 77; // restituire un int quando il tipo di ritorno
                 // non è specificato
29 } // fine della funzione returnImplicitInt
30
31 int demoImplicitInt(x)
32 {
33    return x;
34 } // fine della funzione demoImplicitInt
```

Figura 15.12 Utilizzare implicitamente il tipo int in C89

```
..\examples\ch15\fig15_12.c:5: warning: type defaults to 'int'
in declaration
of 'returnImplicitInt'
..\examples\ch15\fig15_12.c:5: warning: data definition has no type
or storage class
..\examples\ch15\fig15_12.c:6: warning: parameter names (without types)
in function declaration
..\examples\ch15\fig15_12.c:27: warning: return type defaults to 'int'
..\examples\ch15\fig15_12.c:32: warning: return type defaults to 'int'
..\examples\ch15\fig15_12.c: In function 'demoImplicitInt':
..\examples\ch15\fig15_12.c:32: warning: type of "x" defaults to "int"
```

Figura 15.13 Errori di avvertimento per la presenza del tipo implicito int utilizzando l'IDE Dev-C++

15.10 Numeri complessi

Lo standard C99 introduce il supporto per i numeri complessi e l'aritmetica complessa. Il programma riportato in Figura 15.14 effettua le operazioni di base con i numeri complessi.

```
1 // Fig 15.14: fig15_14.c
2 // Utilizzare i numeri complessi in C99
3 #include <stdio.h>
4 #include <complex.h> // per il tipo di dati complesso e le funzioni
                       // matematiche complesse
5
6 int main( void )
7 {
8    double complex a = 32.123 + 24.456 * I ; // a vale 32.123 + 24.456i
9    double complex b = 23.789 + 42.987 * I ; // b vale 23.789 + 42.987i
10   double complex c = 3.0 + 2.0 * I ;
11
12   double complex sum = a + b; // esegue un'addizione complessa
13   double complex pwr = cpow( a, c ); // esegue un elevamento
                                        // a potenza complesso
14
15   printf( "a is %f + %fi\n", creal( a ), cimag( a ));
16   printf( "b is %f + %fi\n", creal( b ), cimag( b ));
17   printf( "a + b is: %f + %fi\n", creal( sum ), cimag( sum ));
18   printf( "a - b is: %f + %fi\n", creal( a - b ), cimag( a - b ));
19   printf( "a * b is: %f + %fi\n", creal( a * b ), cimag( a * b ));
20   printf( "a / b is: %f + %fi\n", creal( a / b ), cimag( a / b ));
21   printf( "a ^ b is: %f + %fi\n", creal( pwr ), cimag( pwr ));
22
23   return 0 ; // indica che il programma è terminato con successo
24 } // fine della funzione main
```

```
a is 32.123000 + 24.456000i
b is 23.789000 + 42.987000i
a + b is: 55.912000 + 67.443000i
a - b is: 8.334000 + -18.531000i
a * b is: -287.116025 + 1962.655185i
a / b is: 0.752119 + -0.331050i
a ^ b is: -17857.051995 + 1365.613958i
```

Figura 15.14 Utilizzare i numeri complessi in C99

Per fare in modo che il C99 riconosca i numeri complessi, è necessario includere il file header `<complex.h>` (riga 4). Ciò permette di espandere la macro `complex` nella parola chiave `_Complex_`: un tipo di dati che riserva un vettore di esattamente due elementi, corrispondenti alle parti reale ed immaginaria di un numero complesso.

Avendo incluso il file header in riga 4, è possibile definire delle variabili come nelle righe 8-10 e 12-13. Ognuna delle variabili `a`, `b`, `c`, `sum` e `pwr` viene definita come variabile di tipo `double complex`. Avremmo potuto usare anche `float complex` oppure `long double complex`.

Gli operatori aritmetici funzionano con i numeri complessi; il file header `<complex.h>` definisce anche diverse funzioni matematiche, per esempio, `cpow` utilizzata in riga 13. [Nota: si possono anche utilizzare gli operatori `!`, `++`, `--`, `&&`, `||`, `==`, `!=` e l'operatore unario `&` con i numeri complessi.]

Le righe 17-21 visualizzano i risultati di varie operazioni aritmetiche. La parte reale e immaginaria di un numero complesso possono essere accedute tramite le funzioni `creal` e `cimag`, rispettivamente, come illustrato nelle righe 15-21. Si noti che nella stringa emessa in output in riga 21, viene utilizzato il simbolo ^ per indicare l'elevamento a potenza.

15.11 Vettori di lunghezza variabile

In C89 i vettori sono di dimensione costante. Cosa succede quando non si conosce la dimensione di un vettore a tempo di compilazione? Per gestire questa situazione, bisogna utilizzare l'allocazione dinamica della memoria con `malloc` e le funzioni correlate. Il C99 permette di gestire i vettori di dimensione sconosciuta utilizzando i vettori di lunghezza variabile (VLA: variable-length array). Questi ultimi non sono vettori la cui dimensione possa cambiare; infatti ciò comprometterebbe l'integrità delle locazioni vicine in memoria. Un *vettore di lunghezza variabile* è un vettore la cui lunghezza o dimensione è definita in termini di un'espressione valutata a tempo di esecuzione. Il programma riportato in Figura 15.15 dichiara e visualizza un VLA.

```
1 // Fig 15.15: fig15_15.c
2 // Utilizzare i vettori di lunghezza variabile in C99
3 #include <stdio.h>
4
5 void printArray( int sz, int arr[ sz ] ); // funzione che accetta un VLA
6
7 int main( void )
8 {
9    int arraySize; // dimensione del vettore
10   printf( "Enter array size in words: " );
11   scanf( "%d", &arraySize );
12
13   // usare un'espressione non costante come dimensione del vettore
14   int array[ arraySize ]; // dichiara un vettore di lunghezza variabile
15   // effettua un test dell'operatore sizeof sul VLA
16   int a = sizeof( array );
17   printf( "\nsizeof yields array size of %d bytes\n\n", a );
18
19   // assegna dei valori agli elementi del VLA
20   for ( int i = 0 ; i < arraySize; i++ ) {
21        array[ i ] = i * i;
22   } // fine del comando for
23
24   printArray( arraySize, array ); // passa il VLA alla funzione
25
26   return 0 ; // indica che il programma è terminato con successo
27 } // fine della funzione main
28
29 void printArray( int size, int array[ size ] )
30 {
31   // visualizza il contenuto del vettore
```

Figura 15.15 Utilizzare i vettori di lunghezza variabile in C99 (continua)

```
32    for ( int i = 0 ; i < size; i++ ) {
33          printf( "array[%d] = %d\n", i, array[ i ] );
34    } // fine del comando for
35 } // fine della funzione printArray
```

```
Enter array size in words: 7
sizeof yields array size of 28 bytes
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25
array[6] = 36
```

Figura 15.15 Utilizzare i vettori di lunghezza variabile in C99

Dopo aver chiesto all'utente la dimensione desiderata (righe 10-11), viene dichiarato un vettore di lunghezza variabile in riga 14. Ciò porta ad un errore di compilazione in C89, ma non in C99 a patto che la variabile che rappresenta la dimensione del vettore sia di tipo intero.

Dopo aver dichiarato il vettore, si utilizza l'operatore `sizeof` in riga 16 per assicurare che il VLA sia della lunghezza appropriata. In C89 `sizeof` è sempre un'operazione svolta a tempo di compilazione, ma quando viene applicata ad un VLA in C99, `sizeof` viene eseguito a tempo di esecuzione. La finestra di output mostra che l'operatore `sizeof` restituisce una dimensione di 28 byte, ovvero, quattro volte quella del numero inserito, in quanto la dimensione di un `int` sul calcolatore utilizzato è 4 byte.

In seguito vengono assegnati dei valori agli elementi del VLA (righe 20-22). Come condizione per la continuazione del ciclo viene utilizzata l'espressione `i < arraySize`. Come con i vettori di lunghezza fissata non c'è nessuna protezione contro un'eventuale fuoriuscita dai limiti del vettore.

Le righe 29-35 definiscono la funzione `printArray` che prende come argomento un VLA. Si noti che la sintassi rimane pressoché la stessa del caso di un normale vettore di lunghezza fissata, ad eccezione del fatto che la dimensione del vettore è una variabile. Anche la variabile utilizzata all'interno delle parentesi del vettore deve essere passata alla funzione. Se usassimo semplicemente la seguente sintassi:

```
void printArray( int array[ size ] )
```

incorreremmo in un errore di compilazione. Se la funzione in oggetto non necessita di conoscere la dimensione del vettore argomento, si può dichiarare il VLA come segue:

```
void printArray( int array[ * ] )
```

che indica il fatto che il chiamante può passare un vettore di qualsiasi lunghezza.

15.12 Altre caratteristiche del C99

In questa sezione verranno fornite delle brevi descrizioni di ulteriori nuove caratteristiche chiave del C99.

Identificatori estesi

Il C89 richiede che le implementazioni del linguaggio supportino degli identificatori di non meno di 31 caratteri per quanto riguarda gli identificatori con collegamento interno (valido soltanto all'interno del file che viene compilato) e di non meno di sei caratteri per quanto riguarda gli identificatori con collegamento esterno (validi anche in altri file). Per maggiori informazioni sul collegamento interno ed esterno si consulti la Sezione 14.5. Lo standard C99 aumenta questi limiti a 63 caratteri per gli identificatori con collegamento interno ed a 31 caratteri per gli identificatori con collegamento esterno. Si noti che questi sono soltanto dei limiti inferiori; i compilatori sono liberi di supportare identificatori con un numero di caratteri maggiore di questi limiti. Gli identificatori ora possono contenere caratteri propri dei linguaggi nazionali tramite gli Universal Character Names (Sezione 6.4.3 dello Standard C99) e, se l'implementazione lo consente, direttamente (Sezione 6.2.4.1 dello Standard C99). [Per ulteriori informazioni si veda la Sezione 5.2.4.1 dello Standard C99.]

La parola chiave restrict

La parola chiave `restrict` viene utilizzata per dichiarare dei puntatori ristretti. Si dichiara un *puntatore ristretto* quando una parte del programma deve avere accesso esclusivo alla regione in memoria acceduta tramite il puntatore. Altri puntatori possono comunque fare riferimento a tale regione della memoria. Tuttavia, le modifiche fatte alla regione della memoria attraverso gli altri puntatori durante l'esistenza del puntatore ristretto vengono perdute. Si può dichiarare un puntatore ristretto ad un int come segue:

```
int *restrict prt;
```

I puntatori ristretti consentono al compilatore di ottimizzare il modo in cui il programma accede alla memoria. Dichiarare scorrettamente un puntatore come ristretto quando un altro puntatore punta alla stessa regione di memoria può provocare un comportamento non definito. [Per ulteriori informazioni si veda la Sezione 6.7.3.1 dello Standard C99.]

Divisione intera affidabile

In C89 il comportamento della divisione intera cambia a seconda dell'implementazione. Alcune implementazioni arrotondano un quoziente negativo verso meno infinito, mentre altre lo arrotondano verso lo zero. Quando uno degli operandi interi è negativo, ciò può provocare dei risultati differenti. Si condideri il caso della divisione di –28 per 5: il risultato esatto è –5,6. Se arrotondiamo il quoziente verso lo zero, otteniamo il risultato intero di –5. Se arrotondiamo –5,6 verso meno infinito, otteniamo un risultato intero di –6. Il C99 elimina tale ambiguità e calcola sempre la divisione intera (ed il resto intero) arrotondando il quoziente verso lo zero. Ciò rende la divisione intera affidabile, in quanto le piattaforme conformi al C99 trattano tutte la divisione intera allo stesso modo. [Per ulteriori informazioni si veda la Sezione 6.5.5 dello Standard C99.]

Membri vettore flessibili

Il C99 consente la dichiarazione di un vettore di lunghezza non specificata come ultimo membro di una `struct`. Si consideri quanto segue

```
struct s {
      int arraySize;
```

```
    int array[];
}; // fine della struct s
```

Un membro vettore flessibile viene dichiarato specificando delle parentesi quadrate vuote (`[]`). Per allocare una `struct` con un membro vettore flessibile, si utilizza del codice come il seguente

```
int desiredSize = 5;
struct s *ptr;
ptr = malloc( sizeof( struct s ) + sizeof( int ) * desiredSize );
```

L'operatore `sizeof` ignora i membri vettore flessibili. L'espressione `sizeof( struct s )` viene valutata come la dimensione di tutti i membri della `struct s` eccezion fatta per il vettore flessibile. Lo spazio in più che viene dichiarato con l'espressione `sizeof( int ) * desiredSize` è la dimensione del vettore flessibile.

Ci sono molte restrizioni sull'utilizzo di membri vettore flessibili. Un membro vettore flessibile può essere dichiarato soltanto come ultimo membro di una `struct` ed ogni `struct` può contenere al più un membro vettore flessibile. Inoltre, un vettore flessibile non può essere l'unico membro di una `struct`: quest'ultima deve avere anche uno o più membri fissati. Qualsiasi `struct` contenente un membro vettore flessibile non può essere un membro di un'altra `struct`. Infine una `struct` con un membro vettore flessibile non può essere inizializzata in modo statico, ma deve essere allocata dinamicamente. Non è possibile determinare la dimensione di un membro vettore flessibile a tempo di compilazione. [Per ulteriori informazioni si veda la Sezione 6.7.2.1 dello Standard C99.]

Tipo di dati long long int

Il C99 introduce il tipo di dati `long long int`, che viene garantito essere lungo almeno 64 bit. I sistemi non dotati di hardware a 64 bit devono emularlo via software. Non è mai considerato equivalente al tipo `long` (anche nel caso sia della stessa dimensione, viene considerato un tipo totalmente diverso). Il C99 introduce il modificatore di lunghezza `ll` (che sta per "`long long`") che può precedere qualsiasi specificatore di conversione intero elencato in Figura 9.1 (ad esempio, `%lld`). [Per ulteriori informazioni si veda la Sezione 6.2.5 dello Standard C99.]

Matematica indipendente dal tipo

Il file header `<tgmath.h>` è nuovo in C99: fornisce delle macro generiche per quanto riguarda i tipi per molte funzioni definite in `<math.h>`. Per esempio, dopo aver incluso `<tgmath.h>`, se `x` è di tipo `float`, l'espressione `sinx(x)` richiamerà `sinf` (la versione `float` di `sin`); se `x` è di tipo `double`, `sin(x)` richiamerà `sin` (che accetta un argomento di tipo `double`); se `x` è di tipo `long double`, `sin(x)` richiamerà `sinl` (la versione `long double` di `sin`); infine se `x` è un numero complesso, `sin(x)` richiamerà la versione appropriata della funzione `sin` per il tipo dei numeri complessi.

Funzioni inline

Il C99 consente la dichiarazione di funzioni inline (come fa il C++) piazzando la parola chiave `inline` prima della dichiarazione della funzione, come nel caso seguente:

```
inline void randomFunction();
```

Ciò non provoca effetti sulla logica del programma dal punto di vista della prospettiva dell'utente, ma può migliorare le prestazioni in quanto le chiamate di funzione richiedono del tempo. Quando si dichiara una funzione inline, il programma non richiamerà più tale funzione. Invece, il compilatore rimpiazzerà ogni chiamata ad una funzione inline con il corpo di tale funzione. Ciò migliora le prestazioni a tempo di esecuzione, ma può incrementare la dimensione del programma. Bisognerebbe dichiarare delle funzioni inline soltanto se sono corte e richiamate frequentemente. La dichiarazione inline costituisce soltanto un consiglio per il compilatore che può decidere di ignorarla. [Per ulteriori informazioni si veda la Sezione 6.7.4 dello Standard C99.]

Istruzioni return senza espressione

Il C99 aggiunge delle restrizioni più consistenti alle istruzioni di ritorno dalle funzioni. Nelle funzioni che restituiscono un valore diverso da void, non è più possibile utilizzare il comando

```
return;
```

In C89 ciò è permesso, ma produce un comportamento non definito se il chiamante tenta di utilizzare il valore restituito dalla funzione. Similmente, nelle funzioni che non restituiscono un valore, non è più possibile restituire un valore. Comandi come i seguenti:

```
void returnInt( ) { return 1 };
```

non sono più consentiti. Il C99 richiede che i compilatori compatibili producano dei messaggi di avvertimento o degli errori di compilazione in ognuno dei casi precedenti. [Per ulteriori informazioni si veda la Sezione 6.8.6.4 dello Standard C99.]

La funzione snprintf: un aiuto nella prevenzione degli attacchi degli hacker

La funzione `snprintf` è nuova in C99 ed è largamente supportata, anche da fornitori che non dichiarano di essere conformi allo standard C99. Risulta di aiuto nella prevenzione dei buffer overflow, una ben nota forma di attacco da parte degli hacker. I prototipi di `sprintf` e `snprintf` sono:

```
int sprintf( char * restrict s, const char * restrict format, … );
int snprintf( char * restrict s, size_t n,
                    const char * restrict format, … );
```

La vecchia funzione `sprintf` scrive in un buffer, ma non sa quanto grande sia tale buffer. Semplicemente scrive quanto è necessario e ciò può danneggiare quanto è allocato dopo il buffer in memoria. La funzione `sprintf` è stata trattata nella tabella in Figura 8.12 e nel codice dell'esempio in Figura 8.15.

Con `snprintf` la dimensione del buffer è passata alla funzione, che non scriverà oltre la fine di quest'ultimo. Ciò preserva l'integrità dei dati in memoria anche in presenza di un errore logico o di un errore nella dimensione del buffer nel programma. Evitando i buffer overflow, `snprintf` rende i programmi più facili da correggere in quanto un buffer overflow può provocare il fallimento di parti scorrelate del programma, rendendo difficile identificare il problema reale. [Per ulteriori informazioni si veda la Sezione 7.19.6.5 dello Standard C99.]

15.13 Risorse disponibili su Internet e sul Web

`www.open-std.org/jtc1/sc22/wg14/`

Sito ufficiale dello standard C99: mette a disposizione i rapporti sui difetti, i documenti di lavoro, i progetti ed i punti di riferimento, la base razionale dello standard C99, i contatti ed altro ancora.

`www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`

Il documento dello Standard C con le revisioni tecniche 1 e 2, ultimo aggiornamento risalente al 6 maggio 2005. (Disponibile gratuitamente).

`www.wiley.com/WileyCDA/WileyTitle/productCd-0470845732.html`

Sito che permette l'acquisto di una copia stampata dello standard C99.

`www.comeaucomputing.com/techtalk/c99/`

Documento contenente le risposte alle domande più comuni sul C99.

`www-128.ibm.com/developerworks/linux/library/l-c99.html?ca=dgr-lnxw961UsingC99`

Articolo: "Sviluppo Open Source con il C99" di Peter Seebach: parla delle caratteristiche della libreria del C99 su sistemi Linux e BSD.

`www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf`

Rapporto ufficiale: "Base razionale per lo Standard Internazionale – Linguaggi di programmazione – C". Questo volume di 224 pagine descrive le delibere del comitato per gli standard relative al C99.

`gcc.gnu.org/c99status.html`

Questo sito web descrive lo stato delle caratteristiche del C99 che sono supportate dalla GNU Compiler Collection (GCC).

`www.kuro5hin.org/story/2001/2/23/194544/139`

Articolo: "Siete pronti per il C99?"; discute alcune delle interessanti nuove caratteristiche, le incompatibilità con il C++ ed il supporto offerto dal compilatore.

`www.informit.com/guides/content.asp?g=cplusplus&seqNum=215&rl=1`

Articolo: "Un Tour del C99" di Danny Kalev; riassume alcune delle nuove caratteristiche dello standard C99.

`www.cuj.com/documents/s=8191/cuj0104meyers/`

Articolo: "Il nuovo C: dichiarazioni e inizializzazioni" di Randy Meyers; discute queste nuove caratteristiche del C99.

`docs.sun.com/source/817-5064/c99.app.html#98079`

Elenca le caratteristiche del C99 supportate dal sistema operativo Solaris.

`home.tiscalinet.ch/t_wolf/tw/c/c9x_changes.html`

Sito che fornisce brevi descrizioni tecniche ed esempi di codice relativi a molte caratteristiche del C99.

`www.bloodshed.net/dev/devcpp.html`

Sito che permette di scaricare Dev-C++, un ambiente integrato gratuito che utilizza la versione MingW del gcc come compilatore per l'utilizzo su sistemi Windows.

`www.digitalmars.com`

Sito che permette di scaricare un compilatore C/C++ per Win32, gratuito e quasi totalmente conforme al C99.

`www.cs.virginia.edu/~lcc-win32/`

Sito che permette di scaricare lcc-win32, un compilatore gratuito quasi totalmente conforme al C99.

`www.openwatcom.org`

Sito che permette di scaricare il compilatore Watcom C/C++, un prodotto gratuito quasi totalmente conforme al C99 della SciTech Software Inc. e della Sybase®.

`developers.sun.com/prodtech/cc/index.jsp`

Sito che permette di scaricare Sun Studio 11, che include un compilatore gratuito completamente conforme al C99.

`david.tribble.com/text/cdiffs.htm`

Articolo: "Incompatibilità fra l'ISO C89 e l'ISO C++" di David Tribble; elenca e descrive le aree in cui ANSI C, C99 e C++ 98 differiscono.

`webstore.ansi.org/ansidocstore/`
`product.asp?sku=INCITS%2FIS0%2FIEC+9899%2D1999`

Sito che permette l'acquisto di una versione elettronica dello Standard C99.

`msdn.microsoft.com/chats/transcripts/vstudio/vstudio_022703.aspx`

Parti di una trascrizione di una chat a cui ha partecipato Brandon Bray, il program manager del compilatore Visual C++. Riporta una discussione sulla futura compatibilità con il C99.

Esercizi di autovalutazione

15.1 Riempite gli spazi vuoti in ognuna delle seguenti frasi:

a) Gli _____ permettono di inizializzare gli elementi di un vettore esplicitamente per mezzo degli indici.
b) Per utilizzare le variabili booleane, includiamo il file header _____ ed usiamo il tipo _____.
c) Un vettore la cui lunghezza sia definita in termini di un valore determinato a tempo di esecuzione è un _____.
d) Gli unici due valori che `Bool` può rappresentare sono _____ e _____.

15.2 Stabilite quali delle seguenti affermazioni siano vere e quali false; per quelle false, spiegate perché lo sono.

a) Il C99 permette di dichiarare delle variabili dopo il loro utilizzo nel programma.
b) Includendo il file header `<complex.h>`, si può utilizzare il tipo di dati `complex`.
c) Tutto il codice valido in C89 è valido anche in C99.
d) Un vettore di lunghezza variabile può cambiare la propria dimensione durante il suo ciclo di vita.

e) Dichiarando una variabile in un comando `for`, non possiamo accedere a quest'ultima al di fuori del comando `for`.
f) `//` indica dove inizia un commento ed un'altra occorrenza di `//` indica dove il commento finisce.

Risposte agli esercizi di autovalutazione

15.1 a) inizializzatori designati. b) `<stdbool.h>`, `bool`. c) vettori di lunghezza variabile. d) `1`, `0`.

15.2 a) Falso. Le variabili devono essere dichiarate prima dell'utilizzo del programma.
b) Vero.
c) Falso. Alcune abitudini, come l'utilizzo del tipo implicito `int`, sono permesse in C89, ma non in C99.
d) Falso. Un vettore di lunghezza variabile ha una dimensione costante una volta che viene dichiarata (a tempo di esecuzione).
e) Vero.
f) Falso. Un'ulteriore occorrenza di `//` sulla stessa riga è trattata come parte del primo commento ed un'altra occorrenza di `//` su una riga differente segna l'inizio di un altro commento.

Esercizi

15.3 Utilizzando i numeri complessi, si scriva un programma che risolva un'equazione di secondo grado della forma $ax^2 + bx + c = 0$ rispetto a x, chiedendo all'utente di inserire i valori reali di a, b e c e visualizzando le due radici sullo schermo. Si ricordi che la soluzione di un'equazione di secondo grado può essere ottenuta utilizzando i calcoli seguenti (illustrati di seguito sotto forma di codice C):

```
X1 = ( -b + csqrt( cpow( b, 2.0 ) - 4.0 * a * c ) ) / ( 2.0 * a );
X2 = ( -b - csqrt( cpow( b, 2.0 ) - 4.0 * a * c ) ) / ( 2.0 * a );
```

15.4 Si utilizzino le dichiarazioni di variabili in C99 nelle intestazioni dei comandi `for` per risolvere il problema seguente. Si prendano in input 20 numeri, ognuno compreso fra 10 e 100, estremi inclusi. Appena ogni numero viene acquisito, lo si visualizzi soltanto se non è un duplicato di un numero già letto. Si pensi al caso peggiore in cui tutti i 20 numeri siano differenti. Si utilizzi il vettore più piccolo possibile per risolvere questo problema.

15.5 Si scriva una funzione `multiple` che determini per una coppia di interi se il secondo sia un multiplo del primo. La funzione dovrebbe prendere come argomenti due interi e dovrebbe restituire `true` nel caso in cui il secondo sia un multiplo del primo e `false` altrimenti. Si utilizzi tale funzione in un programma che prenda in input una serie di coppie di interi.

15.6 Si scriva un programma che calcoli la somma di una sequenza di interi. Si assuma che il primo intero acquisito tramite la `scanf` specifichi il numero di valori rimanenti dal leggere. Si utilizzi un vettore di lunghezza variabile per memorizzare i valori acquisiti in input. Il programma dovrebbe leggere un solo valore ogni volta che `scanf` viene eseguita. Una tipica sequenza di input potrebbe essere

```
5 100 200 300 400 500
```

Appendice A

Risorse su Internet e nel World Wide Web

Questa appendice contiene un elenco di importanti risorse sul linguaggio C disponibili su Internet e nel World Wide Web. Tali risorse comprendono FAQ (Frequently Asked Questions, ovvero, le domande più frequenti), tutorial, informazioni sui compilatori maggiormente usati e su come accedere o ottenere compilatori gratuiti, dimostrazioni, libri, tutorial, strumenti software, articoli, interviste, conferenze, periodici e riviste, corsi online, gruppi di discussioni e risorse per la propria carriera professionale.

Per maggiori informazioni sull'American National Standards Institute (ANSI) o per acquistare documenti su qualche standard, visitate il sito `www.ansi.org`.

A.1 Risorse sul linguaggio C

`www.glenmccl.com/tutor.htm`

Questo sito è un buon punto di riferimento per gli utenti che conoscono il C. Gli argomenti trattati sono corredati da spiegazioni dettagliate e esempi di codice.

`www.programmersheaven.com/zone3/cat155/index.htm`

Questo sito è una notevole risorsa per i programmatori e fornisce molti strumenti per il C.

`www.programmersheaven.com/c/MsgBoard/wwwboard.asp?Board=3`

Questo servizio di messaggi consente agli utenti di inviare domande e commenti sulla programmazione in C e di ricevere risposte dagli altri membri.

`www.codeguru.com/cpp_mfc/index.shtml`

CodeGuru.com è un sito Web molto frequentato dai programmatori e fornisce un elenco molto completo di risorse per chi programma in C.

`www.dinkumware.com/refxc.html`

Il "Dinkum C Library Reference Manual" è un manuale di riferimento scritto da P.J. Plauger ed è interamente disponibile sul Web, fornendo una trattazione completa di tutte le funzioni e le macro della Libreria Standard del C.

`www.thinkage.ca/english/products/index.shtml`

Questo sito Web è pieno di software shareware, tra cui un visualizzatore di codice sorgente ed un parser per l'input.

A.2 FAQ sul linguaggio C

`www.cs.ruu.nl/wais/html/na-dir/C-faq/diff.html`

Questo sito Web contiene gli aggiornamenti ed i cambiamenti relativi alle FAQ del gruppo di discussione `comp.lang.c` (`www.eskimo.com/~scs/C-faq/top.html`).

A.3 Compilatori di programmi C

`ftp://gcc.gnu.org/pub/gcc/releases/index.html`

Un indice completo delle ultime versioni rilasciate del GCC, disponibili gratuitamente.

`www.comeaucomputing.com/features.html`

La Comeau Computing offre gratuitamente il proprio compilatore, che supporta alcune caratteristiche del C99.

`www.compilers.net/`

Compilers.net è un sito progettato per facilitarvi a trovare dei compilatori.

`www.metrowerks.com/MW/Develop/Desktop/Windows/default.htm`

CodeWarrior della Metrowerks è un ambiente di sviluppo per scrivere codice in C/C++ o Java.

`www.faqs.org/faqs/by-newsgroup/comp/comp-compilers.html`

Questo è un sito contenente un elenco delle FAQ generate nel gruppo di discussione `comp.compilers`.

APPENDICE B

Risorse sul C99 disponibili su Internet e nel World Wide Web

Questa appendice fornisce i riferimenti a risorse sul C99, disponibili su Internet e nel World Wide Web, che comprendono FAQ (Frequently Asked Questions, ovvero, le domande più frequenti), tutorial, informazioni su come accedere o ottenere i documenti sullo standard ANSI/ISO C99, dimostrazioni, libri, tutorial, strumenti software, articoli, interviste, conferenze, periodici e riviste, corsi online, gruppi di discussioni e risorse per la propria carriera professionale.

Il C99 costituisce il più recente standard dell'ANSI per il linguaggio di programmazione C. Fu sviluppato come evoluzione del linguaggio C per tenere il passo con il potente hardware odierno e con le richieste degli utenti che andavano man mano crescendo. Lo Standard C99 è maggiormente in grado (rispetto agli standard precedenti del C) di competere con linguaggi come il FORTRAN per quanto riguarda le applicazioni matematiche. Le risorse del C99 comprendono il tipo `long long` per i computer a 64 bit, i numeri complessi per le applicazioni di ingegneria ed un maggior supporto per l'aritmetica a virgola mobile. Inoltre il C99 rende il C maggiormente compatibile con il C++ abilitando il polimorfismo mediante delle funzioni matematiche dai tipi generici e attraverso la creazione di un ben definito tipo booleano.

Lo standard C99 contiene molti cambiamenti rispetto alle versioni precedenti del linguaggio. Questi includono funzionalità avanzate per il calcolo a virgola mobile, i tipi di variabile booleano e `long long`, la rimozione del tipo implicito `int` e la possibilità di dichiarare delle variabili nell'intestazione del ciclo `for`. Delle spiegazioni dettagliate su tutti i cambiamenti apportati nel C99 possono essere reperite nel documento dello standard ANSI/ISO e in molti link sottostanti.

I compilatori conformi al C99 non sono ancora diffusamente disponibili. Alcune fra le librerie del C e i compilatori che supportano il nuovo standard sono la libreria C99 della Dinkumware (`www.dinkumware.com`) e il compilatore C99 della Comeau Computing (`www.comeaucomputing.com`).

Il documento sullo standard internazionale del C99 può essere acquistato dall'American National Standards Institute (`www.ansi.org`). Il documento tecnico che elenca gli errori scoperti nello Standard può essere scaricato gratuitamente. L'InterNational Committee for Information Technology Standards (INCITS, ovvero, Comitato Internazionale per gli Standard dell'Information Technology) svolge il ruolo di Technical Advisory Group (ovvero, Gruppo Consultivo Tecnico) dell'ANSI per l'ISO/IEC Joint Technical Committee 1 (ovvero, Comitato Tecnico Congiunto ISO/IEC 1). La documentazione sul C99 può essere acquistata dal loro sito Web, `www.incits.org`.

B.1 Risorse sul C99

`www.ansi.org`

Tutti i documenti dell'ANSI, incluso lo standard C99, possono essere reperiti ed acquistati da questo sito.

`www.incits.org/tc_home/j11.htm`

Questo sito Web documenta i progressi dell'INCITS (InterNational Committee for Information Technology Standards) nello sviluppo dello standard C.

`anubis.dkuug.dk/JTC1/SC22/WG14/`

L'ISO/IEC JTC1/SC22/WG14 è il gruppo di lavoro internazionale per la standardizzazione del linguaggio di programmazione C. I più recenti aggiornamenti e revisioni apportati al C99 possono essere reperiti in questo sito.

`wwwold.dkuug.dk/JTC1/SC22/WG14/www/newinc9x.htm`

Contiene un elenco delle caratteristiche del C99.

`www.comeaucomputing.com/features.html`

La Comeau Computing offre gratuitamente il proprio compilatore, che supporta alcune caratteristiche del C99.

`www.dinkumware.com/libraries_ref.html`

La Dinkumware concede in licenza delle librerie per C e C++ conformi agli standard ANSI e provviste di documentazione on-line.

`www.thefreecountry.com/compilers/cpp.shtml`

Questo sito Web elenca molti compilatori gratuiti per C e C++, alcuni dei quali si stanno indirizzando verso la compatibilità con il C99.

`david.tribble.com/text/cdiffs.htm`

David R. Tribble discute sulla compatibilità tra il C99 e l'ANSI/ISO C++.

`gcc.gnu.org/c9xstatus.html`

Questo sito Web elenca le più recenti caratteristiche del C99 che sono supportate dalla GNU Compiler Collection (GCC).

`www.cs.ruu.nl/wais/html/na-dir/C-faq/diff.html`

Questo sito Web contiene gli aggiornamenti ed i cambiamenti relativi alle FAQ, che possono essere reperite presso `www.eskimo.com/~scs/C-faq/top.html`, del gruppo di discussione `comp.lang.c`.

`www-ccs.ucsd.edu/c`

Questo sito Web è un esauriente punto di riferimento per la programmazione in Standard C. Comprende e documenta tutte le librerie standard.

`www.lysator.liu.se/c/q8/index.html`

Doug Gwyn fornisce un campionario delle librerie del C99, tutte di pubblico dominio.

`www.ramtex.dk/standard/iostand.htm`

Una proposta di revisioni del C99 relative a problemi di I/O a livello hardware.

`home.att.net/~jackklein/c/standards.html`

Risposte alle FAQ sull'ANSI e sull'ISO e sui motivi per cui gli Standard C e C++ sono importanti.

```
www.cl.cam.ac.uk/~mgk25/c-time/
```

Proposta di una nuova libreria `time` per la più recente bozza del C.

```
www.devworld.apple.com/tools/mpw-tools/c9x.html
```

Questo sito Web riporta il Documento ufficiale del Comitato relativo al C99.

```
www.eskimo.com/~scs/C-faq/top.html
```

Questo elenco di FAQ tratta argomenti come i puntatori, l'allocazione della memoria e le stringhe.

```
comp.lang.c
```

Visitate questo gruppo di discussione per apprendere le ultime "voci di corridoio" sul C99.

```
comp.std.c
```

Questo newsgroup propone una discussione sullo standard C99.

```
gcc.gnu.org/ml/gcc
```

Un gruppo di discussione della GNU che tratta molte questioni, come ad esempio lo standard C99.

APPENDICE C

Tabella di priorità degli operatori

Gli operatori sono mostrati dall'alto in basso in ordine decrescente di priorità.

Operatore del C	Tipo	Associatività
`( )`	parentesi (operatore di chiamata di funzione)	da sinistra a destra
`[ ]`	indice di vettore	
`.`	selezione di un membro tramite oggetto	
`->`	selezione di un membro tramite puntatore	
`++`	operatore unario di preincremento	da destra a sinistra
`--`	operatore unario di predecremento	
`+`	operatore unario più	
`-`	operatore unario meno	
`!`	operatore unario di negazione logica	
`~`	operatore unario di complemento bitwise	
`(` *tipo* `)`	operatore unario di conversione di tipo	
`*`	deriferimento	
`&`	operatore di indirizzo	
`sizeof`	determina la dimensione in byte	
`*`	moltiplicazione	da sinistra a destra
`/`	divisione	
`%`	resto	
`+`	addizione	da sinistra a destra
`-`	sottrazione	
`<<`	scorrimento a sinistra bitwise	da sinistra a destra
`>>`	scorrimento a destra bitwise	
`<`	operatore relazionale minore di	
`<=`	operatore relazionale minore o uguale a	
`>`	operatore relazionale maggiore di	
`>=`	operatore relazionale maggiore o uguale a	

Figura C.1 Tabella di priorità degli operatori in C (continua)

Operatore del C	Tipo	Associatività
==	operatore relazionale di uguaglianza	da sinistra a destra
!=	operatore relazionale di disuguaglianza	
&	AND bitwise	da sinistra a destra
^	OR esclusivo bitwise	da sinistra a destra
|	OR inclusivo bitwise	da sinistra a destra
&&	AND logico	da sinistra a destra
||	OR logico	da sinistra a destra
?:	operatore ternario condizionale	da destra a sinistra
=	operatore di assegnamento	da destra a sinistra
+=	operatore di assegnamento addizione	
-=	operatore di assegnamento sottrazione	
*=	operatore di assegnamento moltiplicazione	
/=	operatore di assegnamento divisione	
%=	operatore di assegnamento resto	
&=	operatore di assegnamento AND bitwise	
^=	operatore di assegnamento OR esclusivo bitwise	
|=	operatore di assegnamento OR inclusivo bitwise	
<<=	operatore di assegnamento scorrimento a sinistra bitwise	
>>=	operatore di assegnamento scorrimento a destra bitwise con segno	
,	virgola	da sinistra a destra

Figura C.1 Tabella di priorità degli operatori in C

Appendice D

L'insieme dei caratteri ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

I numeri a sinistra della tabella rappresentano le cifre più significative del codice del carattere, espresso in notazione decimale (0-127), mentre quelli in cima alla tabella rappresentano le cifre meno significative del codice del carattere. Per esempio, il codice del carattere 'F' è 70, mentre quello di '&' è 38.

Appendice E

I sistemi numerici

Obiettivi

- Apprendere i concetti fondamentali riguardanti i sistemi numerici, come la base, il valore posizionale e quello simbolico.
- Capire come operare con i numeri rappresentati nei sistemi numerici binario, ottale ed esadecimale.
- Imparare ad abbreviare i numeri binari in quelli ottali o esadecimali.
- Imparare a convertire i numeri ottali ed esadecimali in quelli binari.
- Imparare a convertire i numeri decimali nei loro equivalenti binari, ottali ed esadecimali e viceversa.
- Apprendere l'aritmetica binaria e il modo in cui i numeri binari sono rappresentati utilizzando la notazione con complemento a due.

E.1 Introduzione

Questa appendice prenderà in esame i principali sistemi numerici utilizzati dai programmatori C, specialmente da chi si trova a lavorare su progetti software che richiedono una stretta interazione con i componenti hardware a "livello macchina". Progetti di questo tipo comprendono: i sistemi operativi, il software di rete, i compilatori, i sistemi di database e le applicazioni che richiedano prestazioni elevate.

Quando, all'interno di un programma C, scrivete un numero intero come 227 o –63, questo è interpretato come appartenente al *sistema numerico decimale (base 10)*. Le *cifre* del sistema numerico decimale sono 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9, dove 0 è la cifra più bassa mentre 9 è quella più alta (una in meno della base 10). Internamente, però, i computer utilizzano il *sistema numerico binario (base 2)*, che possiede soltanto due cifre, ovverosia 0 e 1; in questo caso, 0 è la cifra più bassa, mentre 1 è quella più alta (una in meno della base 2).

Come vedrete, i numeri binari tendono a essere più lunghi dei loro equivalenti decimali. I programmatori che lavorano con i linguaggi assembly e con quelli di alto livello (come il C), che permettono loro di arrivare fino al "livello macchina", trovano normalmente molto scomodi i numeri binari; è questo il motivo per cui sono diventati popolari due altri sistemi numerici, ovverosia il *sistema numerico ottale (base 8)* e il *sistema numerico esadecimale (base 16)*, che semplificano l'abbreviazione dei numeri binari.

Nel sistema numerico ottale, le cifre vanno dallo 0 al 7; giacché sia il sistema numerico binario sia quello ottale utilizzano meno cifre di quello decimale, queste sono uguali alle loro corrispondenti decimali.

Il sistema numerico esadecimale presenta un problema, giacché richiede sedici cifre (0, quella più bassa, e una più alta con un valore equivalente al 15 decimale, ovverosia uno in meno della base 16). La convenzione vuole che siano utilizzate le lettere dalla A alla F per rappresentare le cifre esadecimali corrispondenti ai valori decimali dal 10 al 15; nel sistema esadecimale, quindi, potranno esservi dei numeri come 876 consistenti solamente di cifre simili a quelle decimali, oppure dei numeri come 8A55F consistenti di cifre e lettere, oppure ancora numeri come FFE consistenti soltanto di lettere. A volte, un numero esadecimale potrà assomigliare a una parola come FACE (volto) o FEED (cibo); tutto ciò potrà sembrare strano ai programmatori abituati a lavorare con i numeri. Le cifre dei sistemi numerici binario, ottale, decimale ed esadecimale sono riassunte nella Figura E.1 e nella Figura E.2.

Ognuno di questi sistemi numerici utilizza la *notazione posizionale*, ovverosia ogni posizione in cui è stata scritta una cifra possiede un diverso *valore posizionale*. Nel caso del numero decimale 937 (il 9, il 3 e il 7 sono definiti *valori simbolo*), per esempio, il 7 è scritto nella posizione delle unità, il 3 in quella delle decine e il 9 in quella delle centinaia. Osservate che ognuna di queste posizioni è una potenza della base (10) e che queste potenze partono da 0 e crescono di 1 man mano che ci si sposta a sinistra all'interno del numero (Figura E.3).

Cifra binaria	Cifra ottale	Cifra decimale	Cifra esadecimale
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valore decimale di10)
			B (valore decimale di 11)
			C (valore decimale di 12)
			D (valore decimale di 13)
			E (valore decimale di 14)
			F (valore decimale di 15)

Figura E.1 Le cifre dei sistemi numerici binario, ottale, decimale ed esadecimale

Attributo	Binario	OttaleDecimale	Esadecimale	
Base	2	8	10	16
Cifra più bassa	0	0	0	0
Cifra più alta	1	7	9	F

Figura E.2 Raffronto tra i sistemi numerici binario, ottale, decimale ed esadecimale

I valori posizionali nel sistema numerico decimale			
Cifra decimale	9	3	7
Nome della posizione	Centinaia	Decine	Unità
Valore posizionale	100	10	1
Valore posizionale espresso come potenza della base (10)	10^2	10^1	10^0

Figura E.3 I valori posizionali nel sistema numerico decimale

Nel caso dei numeri decimali più lunghi, le posizioni successive a sinistra corrisponderebbero alle migliaia (10 alla terza potenza), alle decine di migliaia (10 alla quarta), alle centinaia di migliaia (10 alla quinta), ai milioni (10 alla sesta), alle decine di milioni (10 alla settima), e così via.

Nel numero binario 101, si dice che l'uno all'estrema destra si trova nella posizione degli uno, lo 0 in quella dei due e l'uno all'estrema sinistra in quella dei quattro. Notate che ognuna di queste posizioni è una potenza della base (2), e che queste potenze partono da 0 e aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Figura E.4).

Nel caso dei numeri binari più lunghi, le posizioni successive a sinistra corrisponderebbero a quella degli otto (2 alla terza potenza), a quella dei sedici (2 alla quarta), a quella dei trentadue (2 alla quinta), a quella dei sessantaquattro (2 alla sesta), e così via.

Nel numero ottale 425, si dice che il 5 si trova nella posizione degli uno, il due in quella degli otto e il 4 in quella dei sessantaquattro. Notate che ognuna di queste posizioni è una potenza della base (8), e che queste potenze partono da 0 e aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Figura E.5).

I valori posizionali nel sistema numerico binario			
Cifra binaria	1	0	1
Nome della posizione	Quattro	Due	Uno
Valore posizionale	4	2	1
Valore posizionale espresso come potenza della base (2)	2^2	2^1	2^0

Figura E.4 I valori posizionali nel sistema numerico binario

I valori posizionali nel sistema numerico ottale			
Cifra decimale	4	2	5
Nome della posizione	Sessantaquattro	Otto	Uno
Valore posizionale	64	8	1
Valore posizionale espresso come potenza della base (8)	8^2	8^1	8^0

Figura E.5 I valori posizionali nel sistema numerico ottale

Nel caso dei numeri ottali più lunghi, le posizioni successive a sinistra corrisponderebbero a quella dei cinquecentododici (8 alla terza potenza), a quella dei quattromilanovantasei (8 alla quarta), a quella dei trentaduemilasettecentosessantotto (8 alla quinta), e così via.

Nel numero esadecimale 3DA, si dice che la A si trova nella posizione degli uno, la D in quella dei sedici e il 3 in quella dei duecentocinquantasei. Notate che ognuna di queste posizioni è una potenza della base (16), e che queste potenze partono da 0 e aumentano di 1 man mano che ci si sposta a sinistra all'interno del numero (Figura E.6).

I valori posizionali nel sistema numerico esadecimale

Cifra decimale	3	D	A
Nome della posizione	Duecentocinquantasei	Sedici	Uno
Valore posizionale	256	16	1
Valore posizionale espresso come potenza della base (16)	16^2	16^1	16^0

Figura E.6 I valori posizionali nel sistema numerico esadecimale.

Nel caso dei numeri esadecimali più lunghi, le posizioni successive a sinistra corrisponderebbero a quella dei quattromilanovantasei (16 alla terza potenza), a quella dei sessantacinquemilacinquecentotrentasei (16 alla quarta), e così via.

E.2 L'abbreviazione dei numeri binari in ottali ed esadecimali

Il principale impiego dei numeri ottali ed esadecimali in ambito informatico è l'abbreviazione delle rappresentazioni binarie più lunghe. La Figura E.7 evidenzia che i numeri binari lunghi possono essere espressi in modo più conciso all'interno dei sistemi numerici con basi più alte rispetto a quello binario.

Numero decimale	Rappresentazione binaria	Rappresentazione ottale	Rappresentazione esadecimale
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9

Figura E.7 Gli equivalenti decimali, binari, ottali ed esadecimali (continua)

Numero decimale	Rappresentazione binaria	Rappresentazione ottale	Rappresentazione esadecimale
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Figura E.7 Gli equivalenti decimali, binari, ottali ed esadecimali

Il sistema numerico ottale e quello esadecimale hanno una relazione particolarmente importante con quello binario: le basi degli ottali e degli esadecimali (rispettivamente 8 e 16) sono potenze di quella del sistema numerico binario (base 2). Per esempio, prendete in considerazione il seguente numero binario a 12 cifre con i suoi equivalenti ottale ed esadecimale, e cercate di determinare il modo in cui questa relazione facilita l'abbreviazione dei numeri binari in ottali o esadecimali (la risposta si trova dopo i numeri).

Numero binario	Equivalente ottale	Equivalente esadecimale
100011010001	4321	8D1

Per vedere in che modo un numero binario possa essere facilmente convertito in un ottale, sarà sufficiente suddividere le 12 cifre del numero binario in gruppi di tre bit consecutivi, scrivendo poi questi gruppi al posto delle corrispondenti cifre del numero ottale, così:

```
100   011   010   001
4     3     2     1
```

Notate che la cifra ottale scritta sotto ogni gruppo di tre bit corrisponde precisamente all'equivalente ottale di quel numero binario di 3 cifre, come mostrato nella Figura E.7.

Lo stesso tipo di relazione si potrà osservare nella conversione dei numeri da binari a esadecimali. In particolare, potete provare a suddividere le 12 cifre del numero binario in gruppi di quattro bit consecutivi, scrivendo poi questi gruppi al posto delle corrispondenti cifre del numero esadecimale, così:

```
1000  1101  0001
8     D     1
```

Notate che la cifra esadecimale scritta sotto ogni gruppo di quattro bit corrisponde precisamente all'equivalente esadecimale di quel numero binario di 4 cifre, come mostrato nella Figura E.7.

E.3 La conversione dei numeri ottali ed esadecimali in binari

Nella sezione precedente avete visto come sia possibile convertire i numeri binari nei loro equivalenti ottali ed esadecimali, attraverso la creazione di gruppi di cifre binarie e la loro successiva scrittura nei valori ottali o esadecimali equivalenti. Questo processo potrà essere

utilizzato anche in senso inverso, al fine di produrre l'equivalente binario di un dato numero ottale o esadecimale.

Il numero ottale 653, per esempio, sarà convertito in binario semplicemente scrivendo il 6 nel suo equivalente binario di tre cifre 110, il 5 nel suo equivalente binario 101 e il 3 nel suo equivalente binario 011, ottenendo così il numero binario di nove cifre 110101011.

Il numero esadecimale FAD5, invece, sarà convertito in binario semplicemente scrivendo la F nel suo equivalente binario di 4 cifre 1111, la A nel suo equivalente binario 1010, la D nel suo equivalente binario 1101 e il 5 nel suo equivalente binario 0101, ottenendo così il numero 1111101011010101, formato da 16 cifre.

E.4 La conversione da binario, ottale o esadecimale in decimale

Dato che siamo tutti normalmente abituati a utilizzare i numeri decimali, spesso è più comodo convertire un numero binario, ottale o esadecimale in decimale, al fine di capire meglio il "valore" reale del numero in questione. I diagrammi della sezione E.1 esprimono i valori posizionali in notazione decimale; per convertire un numero in decimale sarà necessario moltiplicare l'equivalente decimale di ogni cifra per il suo valore posizionale e sommare questi prodotti. Il numero binario 110101, per esempio, sarà convertito nel decimale 53 nel modo mostrato dalla Figura E.8.

Convertire un numero binario in decimale

Valori posizionali:	32	16	8	4	2	1
Valori simbolo:	1	1	0	1	0	1
Prodotti:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Somma:	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

Figura E.8 Convertire un numero binario in decimale

Con questa stessa tecnica, per convertire l'ottale 7614 nel decimale 3980 sarà necessario utilizzare i valori posizionali ottali appropriati, come mostrato nella Figura E.9.

Per finire, per convertire il valore esadecimale AD3B nel decimale 44347 sarà necessario utilizzare i valori posizionali esadecimali appropriati, come mostrato nella Figura E.10.

Convertire un numero ottale in decimale

Valori posizionali	512	64	8	1
Valori simbolo:	7	6	1	4
Prodotti:	7*512=3584	6*64=384	1*8=8	4*1=4
Somma:	= 3584 + 384 + 8 + 4 = 3980			

Figura E.9 Convertire un numero ottale in decimale

Convertire un numero esadecimale in decimale

Valori posizionali	4096	256	16	1
Valori simbolo:	A	D	3	B
Prodotti:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Somma:	= 40960 + 3328 + 48 + 11 = 44347			

Figura E.10 Convertire un numero esadecimale in decimale

E.5 La conversione da decimale a binario, ottale o esadecimale

Le conversioni viste nelle sezioni precedenti seguono le convenzioni della notazione posizionale e lo stesso varrà anche per la conversione da decimale a binario, ottale o esadecimale.

Supponendo di voler convertire il valore decimale 57 in binario, sarà necessario partire dalla scrittura dei valori posizionali delle colonne da destra a sinistra, fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi dovrà essere ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

Valori posizionali: 64 32 16 8 4 2 1

Ignorando la colonna con il valore posizionale 64 si otterrà:

Valori posizionali: 32 16 8 4 2 1

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 57 per 32, si otterrà un risultato di 1 con un resto di 25, quindi sarà necessario scrivere 1 nella colonna del 32. Dividendo 25 per 16, si otterrà un risultato di 1 con un resto di 9 e bisognerà quindi scrivere 1 nella colonna del 16. Dividendo 9 per 8, si otterrà un risultato di 1 con un resto di 1. Le due colonne successive produrranno ognuna quozienti uguali a zero quando i loro valori posizionali saranno divisi per 1, quindi sarà necessario scrivere 0 nelle colonne del 4 e del 2. Per finire, 1 diviso 1 farà 1 e quindi sarà necessario scrivere 1 nella colonna dell'uno. Ecco il risultato che si otterrà:

Valori posizionali:	32	16	8	4	2	1
Valori simbolo:	1	1	1	0	0	1

e quindi il valore decimale 57 è equivalente al valore binario 111001.

Per convertire il valore decimale 103 in ottale, bisognerà per prima cosa scrivere i valori posizionali delle colonne fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi sarà ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

Valori posizionali: 512 64 8 1

Ignorando la colonna con il valore posizionale 512 si otterrà:

Valori posizionali: 64 8 1

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 103 per 64, si otterrà un risultato di 1 con un resto di 39 e quindi sarà necessario scrivere 1 nella

colonna del 64. Dividendo 39 per 8, si otterrà un risultato di 4 con un resto di 7 e bisognerà quindi scrivere 4 nella colonna dell'otto. Dividendo 7 per 1, si otterrà un risultato di 7 con un resto di 0 e quindi bisognerà scrivere 7 nella colonna dell'uno. Ecco il risultato che si otterrà:

```
Valori posizionali:  64   8   1
Valori simbolo:      1    4   7
```

e quindi il valore decimale 103 è equivalente al valore ottale 147.

Per convertire il valore decimale 375 in esadecimale, bisognerà per prima cosa scrivere i valori posizionali delle colonne fino a che non sarà stata raggiunta una colonna il cui valore posizionale sia maggiore del numero decimale; questa colonna non sarà necessaria e quindi sarà ignorata. In conformità a quanto appena affermato, ecco cosa bisognerà scrivere:

```
Valori posizionali:  4096  256  16  1
```

Ignorando la colonna con il valore posizionale 4096 si otterrà:

```
Valori posizionali:        256  16  1
```

A questo punto, sarà necessario operare sulle colonne da sinistra verso destra. Dividendo 375 per 256, si otterrà un risultato di 1 con un resto di 119 e quindi sarà necessario scrivere 1 nella colonna del 256. Dividendo 119 per 16, si otterrà un risultato di 7 con un resto di 7 e bisogna quindi scrivere 7 nella colonna del 16. Dividendo 7 per 1, si otterrà un risultato di 7 con un resto di 0 e quindi bisognerà scrivere 7 nella colonna dell'uno. Ecco il risultato che si otterrà:

```
Valori posizionali:  256  16  1
Valori simbolo:      1    7   7
```

e quindi il valore decimale 375 è equivalente al valore esadecimale 177.

E.6 I numeri binari negativi: la notazione con complemento a due

Finora questa appendice ha preso in esame soltanto i numeri positivi; i computer, però, sono in grado di rappresentare anche i numeri negativi, attraverso la *notazione con complemento a due*. Per prima cosa, spiegheremo come si forma il complemento a due di un numero binario, dopodiché passeremo a illustrare in che modo esso rappresenti il valore negativo di un numero binario.

Pensate a una macchina con numeri interi a 32 bit, supponendo che:

```
int value = 13;
```

La rappresentazione a 32 bit di `value` sarà

```
00000000 00000000 00000000 00001101
```

Per formare il negativo di `value`, sarà necessario per prima cosa formare il suo *complemento a uno*, applicando l'operatore C di complemento bitwise (~):

```
onesComplementOfValue = ~value;
```

Internamente, `~value` corrisponderà ora a `value` con tutti i suoi bit invertiti (gli uno saranno diventati zeri e gli zeri saranno diventati uno):

```
value:
00000000 00000000 00000000 00001101

~value (ovvero il complemento a uno di value):
11111111 11111111 11111111 11110010
```

Per formare il complemento a due di `value`, basterà semplicemente aggiungere uno al complemento a uno di `value`:

Complemento a due di value:

```
11111111 11111111 11111111 11110011
```

Ora, se questo fosse effettivamente equivalente a –13, dovremmo essere in grado di aggiungerlo al binario 13 ottenendo un risultato pari a 0:

```
  00000000 00000000 00000000 00001101
+ 11111111 11111111 11111111 11110011
 ---------------------------------
  00000000 00000000 00000000 00000000
```

Il riporto risultante dalla colonna all'estrema sinistra sarà ignorato e quindi si otterrà effettivamente zero come risultato. Aggiungendo il complemento a uno di un numero a quello stesso numero, il risultato sarebbe composto interamente da tanti 1. Il motivo per cui si otterrà un risultato composto solamente da zeri, invece, è che il complemento a due è 1 in più del complemento a uno; l'addizione di questo 1 farà in modo che ogni colonna sommi lo 0 con un riporto di 1, e questo riporto continuerà a essere trasportato a sinistra fino a quando non sarà ignorato nell'ultimo bit, producendo così un risultato di soli zeri.

I computer eseguono in realtà una sottrazione di questo tipo:

```
x = a - value;
```

aggiungendo ad `a` il complemento a due di `value` nel seguente modo:

```
x = a + (~value + 1);
```

Supponete ora che `a` sia 27 e che `value` sia 13; se il complemento a due di `value` fosse realmente il negativo di `value`, aggiungendo ad `a` il complemento a due di `value` si dovrebbe ottenere 14:

```
a (ovverosia 27)   00000000 00000000 00000000 00011011
+ (~value + 1)    +11111111 11111111 11111111 11110011
                  ---------------------------------
                  00000000 00000000 00000000 00001110
```

che equivale effettivamente a 14.

Esercizi di autovalutazione

E.1 Le basi dei sistemi numerici decimale, binario, ottale ed esadecimale sono rispettivamente _____, _____, _____ e _____.

E.2 In generale, le rappresentazioni decimali, ottali ed esadecimali di un dato numero binario contengono (più/meno) cifre rispetto a quelle contenute nel numero binario?

E.3 (*Vero/Falso*) Uno dei motivi per cui è utilizzato il sistema numerico decimale è che permette di abbreviare i numeri binari semplicemente sostituendo una cifra decimale per ogni gruppo di quattro bit binari.

E.4 La rappresentazione (ottale / esadecimale / decimale) di un grande valore binario è la più concisa (tra le alternative proposte).

E.5 (*Vero / Falso*) La cifra più alta in qualsiasi base è una in più della base.

E.6 (*Vero / Falso*) La cifra più bassa in qualsiasi base è una in meno della base.

E.7 Il valore posizionale della cifra all'estrema destra di qualsiasi numero binario, ottale, decimale o esadecimale è sempre _____.

E.8 Il valore posizionale della cifra che si trova alla sinistra della cifra all'estrema destra di un qualsiasi numero binario, ottale, decimale o esadecimale è sempre pari a _____.

E.9 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi numerici indicati:

decimale	1000	100	10	1
esadecimale	...	256	...	...
binario	...	...	...	...
ottale	512	...	8	...

E.10 Convertite il valore binario `110101011000` in ottale e in esadecimale.

E.11 Convertite il valore esadecimale `FACE` in binario.

E.12 Convertite il valore ottale `7316` in binario.

E.13 Convertite il valore esadecimale `4FEC` in ottale. [*Suggerimento*: convertite prima `4FEC` in binario e poi convertite questo valore binario in ottale.]

E.14 Convertite il valore binario `1101110` in decimale.

E.15 Convertite il valore ottale `317` in decimale.

E.16 Convertite il valore esadecimale `EFD4` in decimale.

E.17 Convertite il valore decimale `177` in binario, ottale ed esadecimale.

E.18 Mostrate la rappresentazione binaria del decimale `417`, quindi mostrate il complemento a uno di `417` e il complemento a due di `417`.

E.19 Quale risultato si otterrà aggiungendo il complemento a uno di un numero a quel numero?

Risposte agli esercizi di autovalutazione

E.1 `10`; `2`; `8`; `16`.

E.2 Meno.

E.3 Falso.

E.4 Esadecimale.

E.5 Falso. La cifra più alta in qualunque base è uno meno della base.

E.6 Falso. La cifra più bassa di qualunque base è zero.

E.7 `1` (la base elevata alla potenza zero).

E.8 La base del sistema numerico.

E.9 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi numerici indicati:

decimale	1000	100	10	1
esadecimale	4096	256	16	1
binario	8	4	2	1
ottale	512	64	8	1

E.10 Ottale 6530; Esadecimale D58.

E.11 Binario 1111 1010 1100 1110.

E.12 Binario 111 011 001 110.

E.13 Binario 0 100 111 111 101 100; Ottale 47754.

E.14 Decimale 2+4+8+32+64=110.

E.15 Decimale 7+1*8+3*64=7+8+192=207.

E.16 Decimale 4+13*16+15*256+14*4096=61396.

E.17 Decimale 177
in binario:

```
256    128    64     32     16     8      4      2      1
128    64     32     16     8      4      2      1
(1*128) + (0*64) + (1*32) + (1*16) + (0*8) + (0*4) + (0*2) + (1*1)
10110001
```

in ottale:

```
512    64     8      1
64     8      1
(2*64) + (6*8) + (1*1)
261
```

in esadecimale:

```
256    16     1
16     1
(11*16) + (1*1)
(B*16) + (1*1)
B1
```

E.18 Binario:

```
512    256    128    64     32     16     8      4      2      1
256    128    64     32     16     8      4      2      1
(1*256) + (1*128) + (0*64) + (1*32) + (0*16) + (0*8) + (0*4) + (0*2) + (1*1)
110100001
Complemento a uno:  001011110
Complemento a due:  001011111
Controllo: il numero binario originale + il suo complemento a due
110100001
001011111
- - - -
000000000
```

E.19 Zero.

Esercizi

E.20 Alcune persone sostengono che molti dei nostri calcoli sarebbero più semplici con il sistema numerico in base 12, poiché il 12 è divisibile per molti più numeri rispetto al 10 (del sistema in base 10). Qual è la cifra più bassa della base 12? Quale potrebbe essere il simbolo per la cifra più alta in base 12? Quali sono i valori posizionali delle quattro posizioni all'estrema destra di qualsiasi numero del sistema numerico in base 12?

E.21 Nei sistemi numerici presi in esame in questo capitolo, qual è il rapporto tra il valore del simbolo più alto e quello posizionale della prima cifra che si trova alla sinistra di quella all'estrema destra di ogni numero?

E.22 Completate questa tabella con i valori posizionali relativi alle quattro posizioni all'estrema destra di ognuno dei sistemi numerici indicati:

decimale	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

E.23 Convertite il valore binario 100101111010 in ottale e in esadecimale.

E.24 Convertite il valore esadecimale 3A7D in binario.

E.25 Convertite il valore esadecimale 765F in ottale. [*Suggerimento*: convertite prima 765F in binario e poi convertite il valore binario in ottale.]

E.26 Convertite il valore binario 1011110 in decimale.

E.27 Convertite il valore ottale 426 in decimale.

E.28 Convertite il valore esadecimale FFFF in decimale.

E.29 Convertite il valore decimale 299 in binario, ottale ed esadecimale.

E.30 Mostrate la rappresentazione binaria del decimale 779, quindi mostrate il complemento a uno di 779 e il complemento a due di 779.

E.31 Cosa si otterrà aggiungendo a un numero il suo complemento a due?

E.32 Mostrate il complemento a due del valore intero -1 su una macchina con interi a 32 bit.

Appendice F

Risorse sulla libreria standard del C

Questa appendice fornisce i riferimenti ad importanti risorse sulla libreria standard del C, disponibili su Internet e nel World Wide Web. Le funzioni, i tipi e le macro sono definiti dall'American National Standards Institute e sono progettati per assicurare la portabilità fra i vari sistemi operativi ed incrementare l'efficienza. Anche se non fanno parte del linguaggio C, qualsiasi compilatore che supporti l'ANSI C normalmente fornirà le definizioni di queste librerie.

Nel 1999, l'International Standards Organization ha dato la sua approvazione ad una nuova versione del C, nota come C99. Tale versione supporta molte nuove caratteristiche come specificato nell'Appendice B. Molte delle risorse elencate in seguito forniscono informazioni sulle aggiunte del C99 alla libreria standard del C.

Per ulteriori informazioni sull'ANSI o per acquistare i documenti relativi agli standard, visitate il sito dell'ANSI all'indirizzo `www.ansi.org`.

F.1 Risorse sulla libreria standard del C

`www.ansi.org`

Tutti i documenti dell'ANSI, incluso lo standard C99, possono essere reperiti ed acquistati da questo sito.

`www.incits.org`

L'InterNational Committee for Information Technology Standards (ovvero, il Comitato Internazionale per gli Standard dell'Information Technology) svolge il ruolo di Technical Advisory Group (ovvero, Gruppo Consultivo Tecnico) dell'ANSI per l'ISO/IEC Joint Technical Committee 1 (ovvero, Comitato Tecnico Congiunto ISO/IEC 1). Lo standard C99 può essere reperito ed acquistato in questo sito.

`msdn.microsoft.com/visualc/`

L'home page del Visual C++ contiene collegamenti a molti newsgroup, discussioni e relativi siti nonché informazioni sul supporto ed i miglioramenti di questo prodotto per i linguaggi C/C++.

`www.dinkumware.com/libraries_ref.html`

La Dinkumware concede in licenza delle librerie per C e C++ conformi agli standard ANSI e provviste di documentazione on-line.

`www.cplusplus.com/ref/`

Questo sito Web per la C++ Resources Network (ovvero, la Rete di Risorse per il C++) fornisce i riferimenti per le librerie standard del C++ e del C.

`ccs.ucsd.edu/c/`

Un sito esauriente sullo Standard C messo a disposizione dall'Università della California, San Diego.

`www.infosys.utas.edu.au/info/documentation/C/CStdLib.html`

Materiali di consultazione sullo Standard C messi a disposizione dalla School of Information Systems dell'Università della Tasmania.

`www.acm.uiuc.edu/webmonkeys/book/c_guide`

Manuale di riferimento sulla libreria del C pubblicato dal capitolo dell'Università dell'Illinois dell'Association of Computing Machinery.

`www.thefreecountry.com/compilers/cpp.shtml`

Questo sito Web fornisce librerie per C/C++, editor, IDE (Integrated Development Environment, ovvero, Ambiente di Sviluppo Integrato), compilatori e libri gratuiti.

`www.freeprogrammingresources.com/cpplib.html`

Questo sito Web offre molte risorse gratuite per la programmazione, fra cui librerie per C e C++.

`www.programmersheaven.com/`

Una risorsa completa per programmatori in qualsiasi linguaggio e ambiente. Questo sito offre anche degli strumenti e delle librerie per C e C++.

`www.gnu.org/manual/glibc-2.0.6/html_mono/libc.html`

Un sito Web che descrive lo standard ISO C e tutte le estensioni relative al sistema GNU.

`www.lysator.liu.se/c/rat/title.html`

Questo sito Web presenta la base razionale completa della creazione dell'ANSI C.

`www.lysator.liu.se/c/`

Una collezione di articoli e libri relativi alla storia del C e dello standard ANSI.

P.J. Plauger, in passato Presidente del comitato responsabile per lo sviluppo dello Standard ISO C, ha scritto diversi libri sulle librerie dello Standard C così come su altri argomenti di programmazione. P.J. Plauger fu anche coinvolto nello sviluppo dello Standard C++ e ha pure scritto dei libri sulla libreria di quest'ultimo. Alcuni dei suoi lavori includono:

- *The Standard C Library*, P.J. Plauger, Prentice Hall, 1993-1994.
- *Standard C: A Reference*, P.J. Plauger and Jim Brodie, Prentice Hall, 1996.
- *The Draft Standard C++ Library*, P.J. Plauger, Prentice Hall, 1995.

Appendice G

La libreria standard

G.1 Errori <errno.h>

`EDOM`
`ERANGE`

Questi simboli si espandono in espressioni costanti intere con valori distinti diversi da zero, adatti all'utilizzo nelle direttive `#if` del preprocessore.

`errno`

Un valore di tipo `int` che è impostato con un numero di errore positivo da diverse funzioni della libreria. Il valore di `errno` è zero all'inizio dell'esecuzione del programma, ma non è mai impostato a zero da nessuna funzione della libreria. Un programma che utilizzi `errno` per il controllo dell'errore dovrebbe impostarlo a zero prima della chiamata a una funzione della libreria e controllare il suo valore prima di effettuarne un'altra. All'inizio della propria esecuzione, una funzione della libreria potrebbe salvare il valore di `errno` e impostarlo a zero, ripristinando quello originale, qualora `errno` valga ancora zero poco prima del ritorno. Che ci sia o no un errore, `errno` potrebbe essere impostato a un valore diverso da zero da una chiamata a una funzione della libreria, qualora l'utilizzo di `errno` non sia documentato nella descrizione della funzione riportata dallo standard.

G.2 Definizioni comuni <stddef.h>

`NULL`

Una costante di puntatore nullo definita dall'implementazione.

`offsetof(`*tipo*`,` *designatore-di-membro*`)`

Si espande in un'espressione costante intera di tipo `size_t`, il cui valore è l'offset in byte per il membro della struttura (specificato dal designatore-di-membro) dall'inizio del suo tipo di struttura (specificato da type). Il designatore-di-membro dovrà essere tale che, dato

`static` *tipo* `t;`

l'espressione `&` `(t.`designatore-di-membro`)` sarà valutata in una costante di indirizzo. (Il comportamento sarà indefinito, qualora il membro specificato sia un campo di bit).

`ptrdiff_t`

Il tipo intero con segno restituito dalla sottrazione tra due puntatori.

`size_t`

Il tipo intero senza segno restituito dall'operatore `sizeof`.

`wchar_t`

Un tipo intero, il cui intervallo di valori può rappresentare codici distinti per tutti i membri dell'insieme di caratteri più grande specificato tra quelli delle localizzazioni supportate. Il carattere nullo deve avere il valore di codice zero, mentre ogni membro dell'insieme di caratteri di base deve averne uno uguale a quello che si otterrebbe utilizzandolo da solo in una costante di carattere intera.

G.3 Diagnostica <assert.h>

```
void assert(int espressione);
```

La macro `assert` inserisce delle istruzioni diagnostiche nei programmi. Qualora `espressione` sia falsa, quando sarà eseguita, la macro `assert` scriverà nel file dello standard error delle informazioni relative alla chiamata fallita (incluso il testo dell'argomento, il nome del file sorgente e il suo numero di riga: gli ultimi sono rispettivamente i valori delle macro del preprocessore `__FILE__` e `__LINE__`), in un formato definito dall'implementazione. Il messaggio scritto potrebbe avere la forma

```
Assertion failed: espressione, file xyz, line nnn
```

La macro assert richiamerà quindi la funzione `abort`. Qualora la direttiva del preprocessore

```
#define NDEBUG
```

appaia in un file sorgente in cui sia stato incluso `assert.h`, tutte le asserzioni del file saranno ignorate.

G.4 Gestione dei caratteri <ctype.h>

Le funzioni di questa sezione restituiscono un valore diverso da zero (vero), se e solo se il valore dell'argomento `c` è conforme a quello descritto nel commento della funzione.

```
int isalnum(int c);
```

Verifica qualsiasi carattere per il quale siano vere `isalpha` o `isdigit`.

```
int isalpha(int c);
```

Verifica qualsiasi carattere per il quale siano vere `isupper` o `islower`.

```
int iscntrl(int c);
```

Verifica qualsiasi carattere di controllo.

```
int isdigit(int c);
```

Verifica qualsiasi carattere numerico decimale.

```
int isgraph(int c);
```

Verifica qualsiasi carattere stampabile, eccetto lo spazio `(' ')`.

```
int islower(int c);
```

Verifica qualsiasi carattere che sia una lettera minuscola.

```
int isprint(int c);
```

Verifica qualsiasi carattere stampabile, incluso lo spazio `(' ')`.

```
int ispunct(int c);
```

Verifica ogni carattere stampabile che non sia né uno spazio (`' '`) né un carattere per il quale sia vera `isalnum`.

```
int isspace(int c);
```

Verifica qualsiasi carattere che corrisponda a uno spazio bianco standard. I caratteri di spazio bianco standard sono: lo spazio (`' '`), il salto pagina (`'\f'`), il newline (`'\n'`), il ritorno carrello (`'\r'`), la tabulazione orizzontale (`'\t'`) e quella verticale (`'\v'`).

```
int isupper(int c);
```

Verifica qualsiasi carattere che sia una lettera maiuscola.

```
int isxdigit(int c);
```

Verifica qualsiasi carattere che sia un numero esadecimale.

```
int tolower(int c);
```

Converte una lettera maiuscola nella sua corrispondente minuscola. Se la funzione `isupper`, applicata all'argomento, è vera ed esiste un carattere corrispondente per cui la funzione `islower` sia vera, allora la funzione `tolower` restituirà quest'ultimo, in caso contrario verrà restituito l'argomento inalterato.

```
int toupper(int c);
```

Converte una lettera minuscola nella sua corrispondente maiuscola. Se la funzione `islower`, applicata all'argomento, è vera ed esiste un carattere corrispondente per cui la funzione `isupper` sia vera, allora la funzione `toupper` restituirà quest'ultimo, in caso contrario verrà restituito l'argomento inalterato.

G.5 Localizzazione <locale.h>

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

Questi simboli si espandono in espressioni costanti intere con valori distinti, adatti a essere utilizzati come primo argomento della funzione `setlocale`.

```
NULL
```

Una costante di puntatore nullo definita dall'implementazione.

```
struct lconv
```

Contiene dei membri correlati con la formattazione dei valori numerici. La struttura deve contenere in qualsiasi ordine almeno i seguenti membri. Nella localizzazione `"C"`, i membri devono avere i valori specificati nei commenti.

```
char *decimal_point;        /* "." */
char *thousands-sep;        /* "" */
char *grouping;             /* "" */
char *int_curr_symbol;      /* "" */
```

```
char *currency_simbol        /* "" */
char *mon_decimal_point;     /* "" */
char *mon_thousands_sep;     /* «» */
char *mon_grouping;          /* "" */
char *positive_sign;         /* «» */
char *negative_sign;         /* «» */
char int_frac_digits;        /* CHAR_MAX */
char frac_digits;            /* CHAR_MAX */
char p_cs_precedes;          /* CHAR_MAX */
char p_sep_by_space;         /* CHAR_MAX */
char n_cs_precedes;          /* CHAR_MAX */
char n_sep_by_space;         /* CHAR_MAX */
char p_sign_posn;            /* CHAR_MAX */
char n_sign_posn;            /* CHAR_MAX */
char *setlocale(int categoria, const char *localizzazione);
```

La funzione `setlocale` seleziona la porzione appropriata della localizzazione del programma, come specificato dagli argomenti `categoria` e `localizzazione`. La funzione `setlocale` può essere utilizzata per modificare o interrogare l'intera localizzazione corrente del programma o porzioni di quella. Il valore `LC_ALL` per `categoria` nomina l'intera localizzazione del programma; gli altri valori per `categoria` nominano soltanto una porzione della localizzazione del programma. `LC_COLLATE` influisce sul comportamento delle funzioni `strcoll` e `strxfrm`. `LC_CTYPE` influenza quello delle funzioni per la gestione dei caratteri e di quelle multi-byte. `LC_MONETARY` agisce sulle informazioni relative alla formattazione dei valori monetari restituite dalla funzione `localeconv`. `LC_NUMERIC` agisce sul carattere separatore dei decimali per le funzioni di formattazione dell'input/output, per quelle di conversione delle stringhe e per l'informazione di formattazione dei valori non monetari restituita da `localeconv`. `LC_TIME` influisce sul comportamento di `strftime`.

Un valore `"C"` per `localizzazione` specifica l'ambiente minimo per la traduzione `C`; mentre il valore `""` specifica l'ambiente nativo definito dall'implementazione. Possono essere passate a `setlocale` anche altre stringhe definite dall'implementazione. All'inizio dell'esecuzione del programma sarà eseguita l'equivalente di

```
setlocale(LC_ALL, "C");
```

Qualora per `localizzazione` sia stato fornito un puntatore a una stringa e la selezione possa essere soddisfatta, la funzione `setlocale` restituirà un puntatore alla stringa associata alla `categoria` specificata per la nuova localizzazione. Qualora la selezione non possa essere soddisfatta, la funzione `setlocale` restituirà un puntatore nullo e la localizzazione del programma non sarà cambiata.

Un puntatore nullo per `localizzazione` farà sì che la funzione `setlocale` restituisca un puntatore alla stringa associata alla `categoria` per la localizzazione corrente del programma; questa non sarà modificata.

Il puntatore a stringa restituito dalla funzione `setlocale` è tale che una successiva chiamata con quel valore di stringa e la categoria associata ripristinerà quella parte della localizzazione del programma. La stringa puntata sarà modificabile dal programma, ma potrà essere sostituita da una successiva chiamata alla funzione `setlocale`.

```
struct lconv *localeconv(void);
```

La funzione `localeconv` imposta i componenti di un oggetto di tipo `struct lconv` con i valori appropriati per la formattazione di quantità numeriche (monetarie e non) in accordo con le regole della localizzazione corrente.

I membri di tipo `char *` della struttura sono puntatori a stringhe, ognuno dei quali (eccetto `decimal_point`) può puntare a `""` per indicare che il valore non è disponibile nella localizzazione corrente o è di lunghezza zero. I membri di tipo `char` sono dei numeri non negativi, ognuno dei quali può essere `CHAR_MAX` per indicare che il valore non è disponibile nella localizzazione corrente. I membri inclusi sono:

```
char *decimal_point
```

Il carattere separatore dei decimali, utilizzato per formattare le quantità non monetarie.

```
char *thousands_sep
```

Il carattere utilizzato per separare i gruppi di cifre, prima del carattere separatore dei decimali nelle quantità non monetarie formattate.

```
char *grouping
```

Una stringa i cui elementi indicano la dimensione di ogni gruppo di cifre nelle quantità non monetarie formattate.

```
char *int_curr_symbol
```

Il simbolo internazionale della valuta applicabile alla localizzazione corrente. I primi tre caratteri contengono il simbolo alfabetico internazionale della valuta in accordo con quelli specificati nell'ISO 4217:1987. Il quarto carattere (immediatamente precedente quello nullo) è quello utilizzato per separare il simbolo internazionale della valuta dalla quantità monetaria.

```
char *currency_symbol
```

Il simbolo della valuta della localizzazione applicabile a quella corrente.

```
char *mon_decimal_point
```

Il carattere separatore dei decimali utilizzato per formattare le quantità monetarie.

```
char *mon_thousands_sep
```

Il separatore per i gruppi di cifre prima del carattere separatore dei decimali nelle quantità monetarie formattate.

```
char *mon_grouping
```

Una stringa i cui elementi indicano la dimensione di ogni gruppo di cifre nelle quantità monetarie formattate.

```
char *positive_sign
```

La stringa utilizzata per indicare una quantità monetaria formattata di valore non negativo.

```
char *negative_sign
```

La stringa utilizzata per indicare una quantità monetaria formattata di valore negativo.

```
char int_frac_digits
```

Il numero di cifre frazionarie (quelle dopo il carattere separatore dei decimali) da visualizzare in una quantità monetaria formattata secondo le regole internazionali.

```
char frac_digits
```

Il numero di cifre frazionarie (quelle dopo il carattere separatore dei decimali) da visualizzare in una quantità monetaria formattata.

```
char p_cs_precedes
```

Impostata a 1 o a 0 fa rispettivamente in modo che `currency_symbol` preceda o segua il valore di una quantità monetaria non negativa formattata.

```
char p_sep_by_space
```

Impostata a 1 o a 0 fa rispettivamente in modo che `currency_symbol` sia o no separato con uno spazio dal valore di una quantità monetaria non negativa formattata.

```
char n_cs_precedes
```

Impostata a 1 o a 0 fa rispettivamente in modo che `currency_symbol` preceda o segua il valore di una quantità monetaria negativa formattata.

```
char n_sep_by_space
```

Impostata a 1 o a 0 fa rispettivamente in modo che `currency_symbol` sia o no separato con uno spazio dal valore di una quantità monetaria negativa formattata.

```
char p_sign_posn
```

Impostata a un valore indicante la posizione di `positive_sign` per una quantità monetaria non negativa formattata.

```
char n_sign_posn
```

Impostata a un valore indicante la posizione di `negative_sign` per una quantità monetaria negativa formattata.

Gli elementi di `grouping` e `mon_grouping` sono interpretati in accordo con le seguenti regole:

`CHAR_MAX` Non deve essere eseguito alcun ulteriore raggruppamento.

`0` L'elemento precedente deve essere utilizzato ripetutamente per il resto delle cifre.

altro Il valore intero è il numero di cifre che comprende il gruppo corrente. L'elemento successivo è esaminato per determinare la dimensione del prossimo gruppo di cifre prima di quello corrente.

I valori di `p_sign_posn` e `n_sign_posn` sono interpretati in accordo con le seguenti regole:

`0` Parentesi intorno alla quantità e a `currency_symbol`.

`1` La stringa del segno precede la quantità e il `currency_symbol`.

`2` La stringa del segno segue la quantità e il `currency_symbol`.

3 La stringa del segno precede direttamente il `currency_symbol`.

4 La stringa del segno segue direttamente il `currency_symbol`.

La funzione `localeconv` restituisce un puntatore all'oggetto completato. La struttura puntata dal valore restituito non dovrebbe essere modificata dal programma, ma può essere sostituita da una chiamata susseguente alla funzione `localeconv`. Inoltre, le chiamate alla funzione `setlocale` con le categorie `LC_ALL`, `LC_MONETARY` o `LC_NUMERIC` possono sostituire i contenuti della struttura.

G.6 Matematica <math.h>

`HUGE_VAL`

Una costante simbolica rappresentante un'espressione `double` positiva.

```
double acos(double x);
```

Calcola il valore principale dell'arcocoseno di `x`. Un errore di dominio si verifica con argomenti non compresi nell'intervallo [–1, +1]. La funzione `acos` restituisce l'arcocoseno nell'intervallo [0, π] radianti.

```
double asin(double x);
```

Calcola il valore principale dell'arcoseno di `x`. Un errore di dominio si verifica con argomenti non compresi nell'intervallo [–1, +1]. La funzione `asin` restituisce l'arcoseno nell'intervallo [–π/2, +π/2] radianti.

```
double atan(double x);
```

Calcola il valore principale dell'arcotangente di `x`. La funzione `atan` restituisce l'arcotangente nell'intervallo [–π/2, +π/2] radianti.

```
double atan2(double y, double x);
```

La funzione `atan2` calcola il valore principale dell'arcotangente di `y/x`, usando i segni di entrambi gli argomenti per determinare il quadrante del valore restituito. Un errore di dominio può verificarsi se entrambi gli argomenti sono uguali a zero. La funzione `atan2` restituisce l'arcotangente di `y/x` nell'intervallo [–π, +π] radianti.

```
double cos(double x);
```

Calcola il coseno di `x` (misurato in radianti).

```
double sin(double x);
```

Calcola il seno di `x` (misurato in radianti).

```
double tan(double x);
```

Restituisce la tangente di `x` (misurato in radianti).

```
double cosh(double x);
```

Calcola il coseno iperbolico di `x`. Si verifica un errore di overflow se il valore assoluto di `x` è troppo grande.

```
double sinh(double x);
```

Calcola il seno iperbolico di x. Si verifica un errore di overflow se il valore assoluto di x è troppo grande.

```
double tanh(double x);
```

La funzione tanh calcola la tangente iperbolica di x.

```
double exp(double x);
```

Calcola la funzione esponenziale di x. Si verifica un errore di overflow o di underflow se x è troppo grande o troppo piccolo.

```
double frexp(double valore, int *exp);
```

Suddivide il numero in virgola mobile in una frazione normalizzata e in una potenza intera di 2. Immagazzina l'intero nell'oggetto int puntato da exp. La funzione frexp restituisce il valore x, tale che x è un double con grandezza compresa nell'intervallo [1/2, 1] o zero, mentre valore è uguale a x volte 2 elevato alla potenza *exp. Qualora valore sia zero, entrambe le parti del risultato saranno zero.

```
double ldexp(double x, int exp);
```

Moltiplica un numero in virgola mobile per una potenza intera di 2. Può verificarsi un errore di overflow o di underflow. La funzione ldexp restituisce il valore di x volte 2 elevato alla potenza exp.

```
double log(double x);
```

Calcola il logaritmo naturale di x. Un errore di dominio si verifica se l'argomento è negativo. Un errore di overflow può verificarsi se l'argomento è uguale a zero.

```
double log10(double x);
```

Calcola il logaritmo in base 10 di x. Un errore di dominio si verifica se l'argomento è negativo. Un errore di overflow può verificarsi se l'argomento è uguale a zero.

```
double modf(double valore, double *iptr);
```

Suddivide l'argomento valore in una parte intera e in una frazionaria, ognuna delle quali ha lo stesso segno dell'argomento. Immagazzina la parte intera come double nell'oggetto puntato da iptr. La funzione modf restituisce la parte frazionaria con segno di valore.

```
double pow(double x, double y);
```

Calcola x elevato alla potenza y. Un errore di dominio si verifica se x è negativo e y non è un valore intero. Un errore di dominio si verifica anche se il risultato non può essere rappresentato, quando x è uguale a zero e y è minore o uguale a zero. Può verificarsi un errore di overflow o di underflow.

```
double sqrt(double x);
```

Calcola la radice quadrata non negativa di x. Un errore di dominio si verifica se l'argomento è negativo.

```
double ceil(double x);
```

Calcola il valore intero più piccolo non minore di x.

```
double fabs(double x);
```

Calcola il valore assoluto del numero in virgola mobile `x`.

```
double floor(double x);
```

Calcola il valore intero più grande non maggiore di `x`.

```
double fmod(double x, double y);
```

Calcola il resto in virgola mobile di `x/y`.

G.7 Salti non locali <setjmp.h>

`jmp_buf`

Un tipo vettoriale adatto a mantenere le informazioni necessarie per ripristinare un ambiente chiamante.

```
int setjmp(jmp_buf amb);
```

Salva il suo ambiente chiamante nell'argomento di tipo `jmp_buf`, per un successivo utilizzo da parte della funzione `longjmp`.

Nel caso in cui il ritorno fosse da un'invocazione diretta, la macro `setjmp` restituirebbe il valore zero. Nel caso in cui il ritorno fosse da un'invocazione alla funzione `longjmp`, la macro `setjmp` restituirebbe un valore diverso da zero.

Un'invocazione della macro `setjmp` dovrà apparire solamente in uno dei seguenti contesti:

- l'intera espressione di controllo in un'istruzione di selezione o di iterazione;
- un operando di un operatore relazionale o di uguaglianza il cui secondo operando sia un'espressione intera costante e quella risultante sia l'intera espressione di controllo di un'istruzione di selezione o di iterazione;
- l'operando di un operatore unario ! la cui espressione risultante sia l'intera espressione di controllo di un'istruzione di selezione o di iterazione; o
- l'intera espressione di un'istruzione costituita da una sola espressione.

```
void longjmp(jmp_buf amb, int val);
```

Ripristina l'ambiente salvato dalla più recente invocazione della macro `setjmp` nella stessa chiamata del programma, con il corrispondente argomento di tipo `jmp_buf`. Nel caso in cui non ci fosse stata una tale invocazione, o se la funzione contenente quella della macro `setjmp` avesse nel frattempo terminato la propria esecuzione, il comportamento sarebbe indefinito.

Tutti gli oggetti avranno i valori che avevano nel momento in cui `longjmp` è stata invocata, eccetto quelli degli oggetti con permanenza automatica locali alla funzione contenente l'invocazione della macro `setjmp` corrispondente, che non siano di tipo volatile e che siano stati cambiati tra la chiamata di `setjmp` e quella di `longjmp`; questi ultimi infatti saranno indeterminati.

Poiché aggira il meccanismo usuale di chiamata e ritorno da funzione, `longjmp` dovrà comportarsi correttamente rispetto agli interrupt, ai segnali e ad ogni loro funzione associata. Il comportamento sarebbe tuttavia indefinito, nel caso in cui la funzione `longjmp` dovesse essere invocata da un gestore di segnali nidificato (cioè, da una funzione invocata come risultato di un segnale sollevato durante la gestione di un altro segnale).

Dopo che `longjmp` sarà stata completata, l'esecuzione del programma continuerà come se la corrispondente invocazione della macro `setjmp` avesse appena restituito il valore specificato da `val`. La funzione `longjmp` non potrà fare in modo che la macro `setjmp` restituisca il valore 0; se `val` dovesse essere 0, la macro `setjmp` restituirebbe il valore 1.

G.8 Gestione dei segnali <signal.h>

`sig_atomic_t`

Il tipo intero di un oggetto a cui si può accedere come un'entità atomica, anche in presenza di interrupt asincroni.

`SIG_DFL`
`SIG_ERR`
`SIG_IGN`

Questi simboli si espandono in espressioni costanti con valori distinti, che hanno un tipo compatibile con il secondo argomento e il valore di ritorno della funzione `signal`, e il cui valore è diverso dall'indirizzo di ogni funzione dichiarabile e dalle seguenti costanti, che si espandono in espressioni costanti positive intere corrispondenti al numero del segnale per la condizione specificata:

`SIGABRT`	terminazione anormale, come quella che è avviata dalla funzione `abort`.
`SIGFPE`	un'operazione aritmetica errata, come una divisione per zero o un'operazione risultante in un overflow.
`SIGILL`	individuazione di un'immagine di funzione non valida, come un'istruzione illegale.
`SIGINT`	ricezione di un segnale di attenzione interattivo.
`SIGSEGV`	un accesso non valido alla memoria.
`SIGTERM`	una richiesta di chiusura inviata al programma.

Un'implementazione non ha la necessità di generare nessuno di questi segnali, eccetto che come risultato di chiamate esplicite alla funzione `raise`.

```
void (*signal (int seg, void (*funz)(int)))(int);
```

Sceglie uno dei tre modi nei quali sarà gestito il ricevimento del numero di segnale `seg`. Nel caso in cui il valore di `funz` fosse `SIG_DEF`, per quel segnale sarebbe utilizzata la gestione di default. Nel caso in cui il valore di `funz` fosse `SIG_IGN`, il segnale sarebbe ignorato. Altrimenti, `funz` dovrebbe puntare a una funzione da richiamare qualora il segnale si dovesse presentare. Una tale funzione è detta gestore del segnale.

Qualora `funz` punti a una funzione, quando si presenterà il segnale sarà eseguito per primo l'equivalente di `signal(seg, SIG_DFL);` o un'intercettazione del segnale definita dall'implementazione. (Qualora il valore di `seg` sia `SIGILL` e l'implementazione lo preveda, `SIG_DFL` sarà ripristinato). In seguito sarà eseguito l'equivalente di `(*funz) (seg);`. La funzione `funz` potrebbe terminare eseguendo un'istruzione `return` o richiamando le funzioni `abort`, `exit` o `longjmp`. Qualora `funz` esegua un'istruzione `return` e il valore di `seg` sia `SIGFPE` o qualsiasi altro valore corrispondente a un'eccezione di calcolo, secondo quanto definito dall'implementazione, il comportamento sarà indefinito. Altrimenti, il programma riprenderà l'esecuzione dal punto in cui era stato interrotto.

Qualora si presentasse un segnale diverso da quelli risultanti da chiamate alle funzioni `abort` o `raise`, il comportamento sarebbe indefinito, nel caso che il gestore del segnale richiamasse una funzione della libreria standard che non sia `signal`, passando come primo argomento il numero del segnale corrispondente a quello che ha causato l'invocazione del gestore, o facesse un riferimento a un oggetto statico che non sia l'assegnazione di un valore a una variabile statica del tipo `volatile sig_atomic_t`. Inoltre, se una tale chiamata alla funzione `signal` dovesse risultare in un ritorno `SIG_ERR`, il valore di `errno` sarebbe indeterminato.

All'inizio dell'esecuzione del programma, potrebbe essere eseguito l'equivalente di

```
signal(seg, SIG_IGN);
```

per alcuni segnali selezionati in un modo definito dall'implementazione; mentre l'equivalente di

```
signal(seg, SIG_DFL);
```

sarà eseguito per tutti gli altri segnali definiti dall'implementazione.

Qualora la richiesta possa essere soddisfatta, la funzione `signal` restituirà il valore di `funz` per la chiamata a `signal` più recente per il segnale `seg` specificato. Altrimenti, sarà restituito il valore `SIG_ERR` e in `errno` sarà immagazzinato un valore positivo.

```
int raise(int seg);
```

La funzione `raise` invia il segnale `seg` al programma in esecuzione. La funzione `raise` restituisce zero in caso di successo e un valore diverso da zero in caso di fallimento.

G.9 Argomenti variabili <stdarg.h>

```
va_list
```

Un tipo adatto a mantenere le informazioni necessarie per le macro `va_start`, `va_arg` e `va_end`. Qualora si desideri un accesso ad argomenti variabili, la funzione chiamata dovrà dichiarare un oggetto (chiamato `ap` in questa sezione) di tipo `va_list`. L'oggetto `ap` potrà essere passato come argomento a un'altra funzione; qualora questa invochi la macro `va_arg` con il parametro `ap`, il valore di questo nella funzione chiamante sarà determinato e dovrà essere passato alla macro `va_end`, prima di ogni altro riferimento ad `ap`.

```
void va_start(va_list ap, parmN);
```

Dovrà essere invocata prima di qualsiasi accesso agli argomenti senza nome. La macro `va_start` inizializza `ap` perché `va_arg` e `va_end` possano successivamente utilizzarlo. Il parametro parmN è l'identificatore di quello più a destra nell'elenco variabile di parametri tra quelli inclusi nella definizione di funzione (quello subito prima di `, ...`). Qualora il parametro parmN sia dichiarato con la classe di memoria `register`, con un tipo funzionale o vettoriale, o con uno che non sia compatibile con quello che risulterebbe dopo l'applicazione delle promozioni di default all'argomento, il comportamento sarà indefinito.

```
tipo va_arg(va_list ap, tipo);
```

Si espande in un'espressione che ha il tipo e il valore dell'argomento successivo nella chiamata. Il parametro `ap` dovrà essere lo stesso `va_list ap` inizializzato da `va_start`. Ogni invocazione di `va_arg` modificherà `ap` in modo che siano restituiti a turno i valori degli

argomenti successivi. Il parametro tipo è un nome di tipo specificato in modo tale che quello di un puntatore a un oggetto del tipo specificato possa essere ottenuto semplicemente aggiungendo il suffisso * a tipo. Qualora non ci sia nessun argomento successivo, o qualora il tipo non sia compatibile con quello dell'argomento successivo (una volta promosso in base alle promozioni di default), il comportamento sarà indefinito. La prima invocazione della macro `va_arg`, dopo quella di `va_start`, restituirà il valore dell'argomento successivo a quello specificato da parmN. Le invocazioni susseguenti restituiranno in successione i valori degli argomenti rimanenti.

```
void va_end(va_list ap);
```

Facilita un normale ritorno da una funzione il cui elenco variabile di argomenti sia stato oggetto di riferimento da parte dell'espansione con cui `va_start` ha inizializzato `va_list ap`. La macro `va_end` può modificare `ap` in modo che non sia più utilizzabile (senza l'intervento di una chiamata a `va_start`). Qualora non ci sia un'invocazione corrispondente della macro `va_start`, o qualora `va_end` non sia stata invocata prima del ritorno, il comportamento sarà indefinito.

G.10 Input/Output <stdio.h>

```
_IOFBF
_IOLBF
_IONBF
```

Espressioni costanti intere con valori distinti, adatte all'utilizzo come terzo argomento della funzione `setvbuf`.

```
BUFSIZ
```

Un'espressione costante intera, che rappresenta la dimensione del buffer utilizzato dalla funzione `setbuf`.

```
EOF
```

Un'espressione costante intera negativa restituita da diverse funzioni per indicare la fine del file, ovverosia, che non c'è più alcun input nello stream.

```
FILE
```

Un tipo di oggetto in grado di registrare tutte le informazioni necessarie per controllare uno stream, incluso il suo indicatore di posizione del file, un puntatore al buffer associato (qualora ce ne sia uno), un indicatore di errore che registri se si verifichino errori in lettura/scrittura, e un indicatore di fine del file che registri se sia stata raggiunta la fine del medesimo.

```
FILENAME_MAX
```

Un'espressione costante intera corrispondente alla dimensione di un vettore di tipo `char`, che sia sufficientemente grande per contenere la stringa del nome di file più lungo che l'implementazione garantisca di poter aprire.

```
FOPEN_MAX
```

Un'espressione costante intera corrispondente al numero minimo di file che l'implementazione garantisca di poter aprire contemporaneamente.

```
fpos_t
```

Un tipo di oggetto in grado di registrare tutte le informazioni necessarie per specificare in modo univoco ogni posizione all'interno di un file.

`L_tmpnam`

Un'espressione costante intera corrispondente alla dimensione di un vettore di tipo `char`, che sia sufficientemente grande per contenere la stringa del nome di un file temporaneo generata dalla funzione `tmpnam`.

`NULL`

Una costante di tipo puntatore nullo definita dall'implementazione.

`SEEK_CUR`
`SEEK_END`
`SEEK_SET`

Espressioni costanti intere con valori distinti, adatte all'utilizzo come terzo argomento della funzione `fseek`.

`size_t`

Il tipo intero senza segno restituito dall'operatore `sizeof`.

`stderr`

Espressione di tipo "puntatore a `FILE`" che fa riferimento all'oggetto `FILE` associato allo stream dello standard error.

`stdin`

Espressione di tipo "puntatore a `FILE`" che fa riferimento all'oggetto `FILE` associato allo stream dello standard input.

`stdout`

Espressione di tipo "puntatore a `FILE`" che fa riferimento all'oggetto `FILE` associato allo stream dello standard output.

`TMP_MAX`

Un'espressione costante intera corrispondente al numero minimo di nomi di file univoci che possano essere generati dalla funzione `tmpnam`. Il valore della macro `TMP_MAX` deve essere almeno 25.

```
int remove(const char *nomefile);
```

Fa in modo che il file il cui nome è dato dalla stringa puntata da `nomefile` non sia più accessibile con quel nome. Un tentativo susseguente di aprire quel file utilizzando quel nome fallirà, a meno che non sia stato creato nuovamente. Il comportamento della funzione `remove`, nel caso che il file sia correntemente aperto, dipende dall'implementazione. La funzione `remove` restituirà zero qualora l'operazione abbia successo, un valore diverso da zero qualora fallisca.

```
int rename(const char *vecchio, const char *nuovo);
```

Fa in modo che il file il cui nome corrisponde alla stringa puntata da `vecchio` sia rinominato, da questo momento in poi, con il nome contenuto nella stringa puntata da `nuovo`. Il file chiamato `vecchio` non sarà più accessibile con quel nome. Il comportamento della funzione

rename, nel caso che un file chiamato con la stringa puntata da nuovo esista già prima della sua invocazione, dipenderà dall'implementazione. La funzione rename restituirà zero qualora l'operazione abbia successo, un valore diverso da zero qualora fallisca; in questo caso e qualora il file esistesse già, conserverebbe ancora il suo nome originale.

```
FILE *tmpfile(void);
```

Crea un file binario temporaneo che sarà rimosso automaticamente alla sua chiusura o alla fine del programma. L'eventuale rimozione di un file temporaneo aperto, nel caso che il programma termini in modo anormale, dipenderà dall'implementazione. Il file sarà aperto per l'aggiornamento con il modo "wb+". La funzione tmpfile restituirà un puntatore allo stream del file che sarà stato creato. Qualora il file non possa essere creato, la funzione tmpfile restituirà un puntatore nullo.

```
char *tmpnam(char *s);
```

La funzione tmpnam genera una stringa che sarà un nome di file valido e che sarà diverso da quelli di qualsiasi file esistente. La funzione tmpnam genererà una stringa differente ogni volta che sarà richiamata, fino a un massimo di TMP_MAX volte. Il suo comportamento, qualora sia richiamata più di TMP_MAX volte, dipenderà dall'implementazione.

Qualora l'argomento sia un puntatore nullo, la funzione tmpnam lascerà il suo risultato in un oggetto statico interno e ne restituirà un puntatore. Le successive chiamate della funzione tmpnam potranno modificare lo stesso oggetto. Qualora l'argomento non sia un puntatore nullo, si assumerà che punti a un vettore di almeno L_tmpnam caratteri; la funzione tmpnam scriverà il proprio risultato in quel vettore e restituirà l'argomento.

```
int fclose(FILE *stream);
```

La funzione fclose fa in modo che lo stream puntato da stream sia svuotato e che il file associato sia chiuso. Ogni dato presente nel buffer e non ancora scritto per quello stream sarà inviato all'ambiente di esecuzione perché sia scritto nel file; ogni dato presente nel buffer e non ancora letto sarà dimenticato. Lo stream sarà dissociato dal file. Qualora il buffer associato sia stato allocato in modo automatico, sarà rilasciato. La funzione fclose restituirà zero qualora lo stream sia stato chiuso con successo, o EOF nel caso che sia stato rilevato un errore qualsiasi.

```
int fflush(FILE *stream);
```

Nel caso che stream punti a uno stream di output o di aggiornamento in cui l'operazione più recente non sia stata una di input, la funzione fflush farà in modo che i dati non ancora scritti per quello stream siano inviati all'ambiente di esecuzione perché siano scritti nel file; altrimenti, il comportamento sarà indefinito.

Nel caso che stream sia un puntatore nullo, la funzione fflush eseguirà la suddetta azione di svuotamento su tutti gli stream che abbiano le suddette caratteristiche. La funzione fflush restituirà EOF nel caso che si verifichi un errore di scrittura, zero in caso contrario.

```
FILE *fopen(const char *nomefile, const char *modo);
```

La funzione fopen apre il file il cui nome corrisponde alla stringa puntata da nomefile e vi associa uno stream. L'argomento modo punta a una stringa che incomincia con una delle seguenti sequenze:

`r`	apre un file di testo per la lettura
`w`	tronca la lunghezza a zero o crea un file di testo per la scrittura
`a`	accodamento; apre o crea un file di testo per scrivere alla fine dello stesso
`rb`	apre un file binario per la lettura
`wb`	tronca la lunghezza a zero o crea un file binario per la scrittura
`ab`	accodamento; apre o crea un file binario per la scrittura alla fine dello stesso
`r+`	apre un file di testo per l'aggiornamento (lettura e scrittura)
`w+`	tronca la lunghezza a zero o crea un file di testo per l'aggiornamento
`a+`	accodamento; apre o crea un file di testo per l'aggiornamento, scrivendo alla fine dello stesso
`r+b` o `rb+`	apre un file binario per l'aggiornamento (lettura e scrittura)
`w+b` o `wb+`	tronca la lunghezza a zero o crea un file binario per l'aggiornamento
`a+b` o `ab+`	accodamento; apre o crea un file binario per l'aggiornamento, scrivendo alla fine dello stesso

L'apertura di un file con il modo di lettura (`'r'` come primo carattere nell'argomento `modo`) fallirà qualora il file non esista o non possa essere letto. L'apertura del file con il modo di accodamento (`'a'` come primo carattere nell'argomento `modo`) farà in modo che tutte le successive scritture nel file siano forzate alla sua fine corrente, ignorando eventuali chiamate alla funzione `fseek`. In alcune implementazioni, aprire un file binario con il modo di accodamento (`'b'` come secondo o terzo carattere nella succitata lista di valori per l'argomento `modo`) potrà inizialmente posizionare l'indicatore per lo stream oltre l'ultimo dato scritto, a causa del riempimento dello spazio rimanente con caratteri nulli.

Quando un file sarà aperto con il modo di aggiornamento (`'+'` come secondo o terzo carattere nella succitata lista di valori per l'argomento `modo`), sullo stream associato potranno essere eseguite delle operazioni di input e di output. Tuttavia, l'output potrebbe non essere seguito direttamente da un input senza la frapposizione di una chiamata alla funzione `fflush` o a quelle di posizionamento nel file (`fseek`, `fsetpos` o `rewind`), e l'input potrebbe non essere seguito direttamente dall'output senza la frapposizione di una chiamata alle funzioni di posizionamento nel file, sempre che l'operazione di input non incontri la fine del file. In alcune implementazioni, aprire (o creare) un file di testo con il modo di aggiornamento potrebbe invece aprire (o creare) uno stream binario.

Quando sarà aperto, uno stream sarà gestito completamente con una memoria tampone (buffer) se e solo se potrà essere determinato che non faccia riferimento a un dispositivo interattivo. Gli indicatori di errore e di fine del file per lo stream saranno azzerati. La funzione `fopen` restituirà un puntatore all'oggetto che controllerà lo stream. Qualora l'operazione di apertura fallisse, `fopen` restituirebbe un puntatore nullo.

```
FILE *freopen(const char *nomefile, const char *modo, FILE *stream);
```

La funzione `freopen` apre il file il cui nome corrisponde alla stringa puntata da `nomefile` e vi associa lo stream puntato da `stream`. L'argomento `modo` è utilizzato proprio come nella funzione `fopen`.

La funzione `freopen` tenterà prima di chiudere qualsiasi file associato allo stream specificato. Un eventuale fallimento della chiusura del file sarà ignorato. Gli indicatori di errore e

di fine del file per lo stream saranno azzerati. La funzione `freopen` restituirà un puntatore nullo qualora l'operazione di apertura fallisca. Altrimenti, `freopen` restituirà il valore di `stream`.

```
void setbuf(FILE *stream, char *buf);
```

La funzione `setbuf` è equivalente a `setvbuf` invocata con i valori `_IOFBF` per `modo` e `BUFSIZ` per `dimensione`, o (qualora `buf` sia un puntatore nullo), con il valore `_IONBF` per `modo`. La funzione `setbuf` non restituisce alcun valore.

```
int setvbuf(FILE *stream, char *buf, int modo, size_t dimensione);
```

La funzione `setvbuf` può essere utilizzata soltanto dopo che lo stream puntato da `stream` sarà stato associato a un file aperto, e prima che sia eseguita qualsiasi altra operazione sullo stream. L'argomento `modo` determinerà il modo in cui sarà gestito il buffer di `stream`: `_IOFBF` farà in modo che l'input/output passi totalmente dal buffer; `_IOLBF` farà in modo che l'input/output sia passato nel buffer per righe; `_IONBF` farà in modo che l'input/output non passi da un buffer. Qualora `buf` non sia un puntatore nullo, il vettore cui fa riferimento potrà essere utilizzato in sostituzione del buffer allocato dalla funzione `setvbuf`. L'argomento `dimensione` specifica quella del vettore. Il contenuto del vettore sarà sempre indeterminato. La funzione `setvbuf` restituirà zero in caso di successo, o un valore diverso da zero qualora a `modo` sia stato assegnato un valore non valido, o qualora la richiesta non possa essere soddisfatta.

```
int fprintf(FILE *stream, const char *formato, ...);
```

La funzione `fprintf` invia il proprio output nello stream puntato da `stream`, in modo controllato dalla stringa puntata da `formato`, che specificherà in che modo debbano essere convertiti per l'output gli argomenti successivi. Qualora non ci siano argomenti sufficienti per il formato, il comportamento sarà indefinito. Qualora il formato sia stato esaurito mentre rimangono ancora degli argomenti, quelli in eccesso saranno come sempre valutati, ma per il resto saranno ignorati. La funzione `fprintf` restituirà il controllo quando avrà incontrato la fine della stringa del formato. Consultate il Capitolo 9, "L'input/output formattato", per una descrizione dettagliata delle specifiche per la conversione dell'output. La funzione `fprintf` restituirà il numero di caratteri inviati in output, o un valore negativo qualora si verifichi un errore.

```
int fscanf(FILE *stream, const char *formato, ...);
```

La funzione `fscanf` legge l'input dallo stream puntato da `stream`, in modo controllato dalla stringa puntata da `formato`, che specificherà le sequenze di input ammissibili e il modo in cui queste dovranno essere convertite per l'assegnamento, utilizzando gli argomenti successivi come puntatori agli oggetti che riceveranno l'input convertito. Qualora non ci siano argomenti sufficienti per il formato, il comportamento sarà indefinito. Qualora il formato sia stato esaurito mentre rimangono ancora degli argomenti, quelli in eccesso saranno come sempre valutati, ma per il resto saranno ignorati. Consultate il Capitolo 9, "L'input/output formattato", per una descrizione dettagliata delle specifiche per la conversione dell'input.

La funzione `fscanf` restituirà il valore della macro `EOF` qualora si verifichi un errore di input prima di qualsiasi conversione. Altrimenti, la funzione `fscanf` restituirà il numero degli elementi assegnati, che potranno anche essere inferiori a quelli forniti, o anche zero, in caso di un prematuro fallimento di corrispondenza.

```
int printf(const char *formato, ...);
```

La funzione `printf` è equivalente a `fprintf` con l'argomento `stdout` inserito prima dei rimanenti. La funzione `printf` restituirà il numero di caratteri inviati in output, o un valore negativo qualora si sia verificato un errore.

```
int scanf(const char *formato, ...);
```

La funzione `scanf` è equivalente a `fscanf` con l'argomento `stdin` inserito prima dei rimanenti. La funzione `scanf` restituirà il valore della macro `EOF`, qualora si sia verificato un errore di input prima di qualsiasi conversione. Altrimenti, `scanf` restituirà il numero degli elementi assegnati, che potranno anche essere inferiori a quelli forniti, o anche zero, in caso di un prematuro fallimento di corrispondenza.

```
int sprintf(char *s, const char *formato, ...);
```

La funzione `sprintf` è equivalente a `fprintf`, eccetto che l'argomento `s` specifica un vettore nel quale l'output generato dovrà essere inviato, invece di specificare uno stream. Un carattere nullo sarà accodato alla fine di quelli inviati in output, ma non sarà contato nella somma restituita. Il comportamento della copia fra oggetti che si sovrappongano sarà indefinito. La funzione `sprintf` restituirà il numero di caratteri scritti nel vettore, escluso quello nullo di terminazione.

```
int sscanf(const char *s, const char *formato, ...);
```

La funzione `sscanf` è equivalente a `fscanf`, eccetto che l'argomento `s` specifica una stringa dalla quale dovrà essere ottenuto l'input, invece di specificare uno stream. Il raggiungimento della fine della stringa sarà equivalente alla fine del file incontrata dalla funzione `fscanf`. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito.

La funzione `sscanf` restituirà il valore della macro `EOF`, qualora si sia verificato un errore di input prima di qualsiasi conversione. Altrimenti, `sscanf` restituirà il numero degli elementi assegnati, che potranno anche essere inferiori a quelli forniti, o anche zero, in caso di un prematuro fallimento di corrispondenza.

```
int vfprintf(FILE *stream, const char *formato, va_list arg);
```

La funzione `vfprintf` è equivalente a una `fprintf`, in cui l'elenco variabile di argomenti sia stato sostituito da un `arg` inizializzato dalla macro `va_start` (e dalle probabili chiamate successive a `va_arg`). La funzione `vfprintf` non invocherà la macro `va_end`. Essa restituirà il numero dei caratteri inviati in output, o un valore negativo qualora si sia verificato un errore.

```
int vprintf(const char *formato, va_list arg);
```

La funzione `vprintf` è equivalente a una `printf`, in cui l'elenco variabile di argomenti sia stato sostituito da un `arg` che dovrebbe essere stato inizializzato dalla macro `va_start` (e dalle probabili chiamate successive a `va_arg`). La funzione `vprintf` non invocherà la macro `va_end`. Essa restituirà il numero dei caratteri inviati in output, o un valore negativo qualora si sia verificato un errore.

```
int vsprintf(char *s, const char *formato, va_list arg);
```

La funzione `vsprintf` è equivalente a una `sprintf`, in cui l'elenco variabile di argomenti sia stato sostituito da un `arg` che dovrebbe essere stato inizializzato dalla macro `va_start` (e dalle probabili chiamate successive a `va_arg`). La funzione `vsprintf` non invocherà la

macro va_end. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione vsprintf restituirà il numero dei caratteri scritti nel vettore, escluso quello nullo di terminazione.

```
int fgetc(FILE *stream);
```

La funzione fgetc ottiene dallo stream di input puntato da stream il carattere successivo (se presente), come un unsigned char convertito in int, e fa avanzare (se definito) l'indicatore di posizione del file associato allo stream. La funzione fgetc restituirà il carattere successivo dello stream di input puntato da stream. Qualora lo stream sia alla fine del file, sarà impostato il relativo indicatore e fgetc restituirà un EOF. Qualora si verifichi un errore di lettura, sarà impostato il relativo indicatore dello stream e fgetc restituirà un EOF.

```
char *fgets(char *s, int n, FILE *stream);
```

La funzione fgets legge dallo stream puntato da stream un numero massimo di caratteri inferiore di un'unità a quanto specificato da n, immagazzinandoli nel vettore puntato da s. Non saranno letti ulteriori caratteri dopo un newline (che sarà conservato), o dopo la fine del file. Immediatamente dopo l'ultimo carattere letto, nel vettore sarà inserito quello nullo di terminazione.

La funzione fgets restituirà s in caso di successo. Qualora sia stata incontrata la fine del file e nessun carattere sia stato letto e immagazzinato nel vettore, i suoi contenuti resteranno invariati e sarà restituito un puntatore nullo. Qualora durante l'operazione si verifichi un errore di lettura, i contenuti del vettore saranno indeterminati e sarà restituito un puntatore nullo.

```
int fputc(int c, FILE *stream);
```

La funzione fputc scrive il carattere specificato da c (convertito in un unsigned char) nello stream di output puntato da stream, inserendolo nel posto individuato dall'indicatore di posizione del file associato allo stream (qualora sia stato definito), e facendo avanzare in modo appropriato il suddetto indicatore. Qualora il file non possa supportare le richieste di posizionamento, o nel caso che lo stream sia stato aperto in accodamento, il carattere sarà accodato allo stream di output. La funzione fputc restituirà il carattere scritto. Qualora si verifichi un errore di scrittura, sarà impostato il relativo indicatore dello stream e fputc restituirà un EOF.

```
int fputs(const char *s, FILE *stream);
```

La funzione fputs scrive la stringa puntata da s nello stream puntato da stream. Il carattere nullo di terminazione non sarà scritto. La funzione fputs restituirà un EOF qualora si verifichi un errore di scrittura; altrimenti restituirà un valore non negativo.

```
int getc(FILE *stream);
```

La funzione getc è equivalente a fgetc eccetto che, qualora sia stata implementata con una macro, potrebbe valutare stream più di una volta. Di conseguenza, l'argomento passato dovrà essere un'espressione che non produca effetti collaterali.

La funzione getc restituirà il carattere successivo dello stream di input puntato da stream. Qualora questo sia alla fine del file, sarà impostato il relativo indicatore dello stream e getc restituirà un EOF. Qualora si verifichi un errore di lettura, sarà impostato il relativo indicatore dello stream e getc restituirà un EOF.

```
int getchar(void);
```

La funzione `getchar` è equivalente a un `getc` con l'argomento `stdin`. La funzione `getchar` restituirà il carattere successivo dello stream di input puntato da `stdin`. Qualora questo sia alla fine del file, sarà impostato il relativo indicatore dello stream e `getchar` restituirà un `EOF`. Qualora si verifichi un errore di lettura, sarà impostato il relativo indicatore dello stream e `getchar` restituirà un `EOF`.

```
char *gets(char *s);
```

La funzione `gets` legge dei caratteri dallo stream di input puntato da `stdin` e li immagazzina nel vettore puntato da `s`, finché non incontra la fine del file o non legge un carattere newline. Questi saranno ignorati e nel vettore, immediatamente dopo l'ultimo carattere letto, sarà inserito quello nullo di terminazione. La funzione `gets` restituirà `s` in caso di successo. Qualora sia stata incontrata la fine del file e nessun carattere sia stato letto e immagazzinato nel vettore, i suoi contenuti resteranno invariati e sarà restituito un puntatore nullo. Qualora si verifichi un errore di lettura durante l'operazione, i contenuti del vettore saranno indeterminati e sarà restituito un puntatore nullo.

```
int putc(int c, FILE *stream);
```

La funzione `putc` è equivalente a `fputc` eccetto che, qualora sia stata implementata con una macro, potrebbe valutare `stream` più di una volta. Di conseguenza l'argomento non dovrà mai essere un'espressione che produca effetti collaterali. La funzione `putc` restituirà il carattere scritto. Qualora si verifichi un errore di scrittura, sarà impostato il relativo indicatore dello stream e `putc` restituirà un `EOF`.

```
int putchar(int c);
```

La funzione `putchar` è equivalente a un `putc` con il secondo argomento `stdout`. La funzione `putchar` restituirà il carattere scritto. Qualora si verifichi un errore di scrittura, sarà impostato il relativo indicatore dello stream e `putchar` restituirà un `EOF`.

```
int puts(const char *s);
```

La funzione `puts` scrive la stringa puntata da `s` nello stream puntato da `stdout`, accodando all'output un carattere newline. Il carattere nullo di terminazione non sarà scritto. La funzione `puts` restituirà un `EOF` qualora si verifichi un errore di scrittura; altrimenti restituirà un valore non negativo.

```
int ungetc(int c, FILE *stream);
```

La funzione `ungetc` rinvia il carattere specificato da `c` (convertito in un `unsigned char`) nello stream di input puntato da `stream`. I caratteri rinviati saranno restituiti in ordine inverso dalle successive letture su quello stream. La frapposizione di una chiamata con successo (sullo stream puntato da `stream`) di una funzione di posizionamento nel file (`fseek`, `fsetpos` o `rewind`) eliminerà tutti i caratteri rinviati allo stream. La memoria esterna corrispondente allo stream resterà immutata.

Il rinvio di un singolo carattere è sempre garantito. Nel caso in cui la funzione `ungetc` sia invocata troppe volte sullo stesso stream, senza frapporre delle operazioni di lettura o di riposizionamento nel file, ci potrebbe essere un fallimento. Se il valore di `c` dovesse essere uguale a `EOF`, la funzione fallirebbe e lo stream di input resterebbe invariato.

Una chiamata con successo alla funzione `ungetc` azzererà l'indicatore di fine file dello stream. Dopo la lettura e l'eliminazione di tutti i caratteri rinviati, il valore dell'indicatore di posizione per il file dello stream dovrebbe corrispondere a quello precedente il rinvio dei

caratteri. Per uno stream di testo, il valore del suo indicatore di posizione del file, dopo una chiamata con successo alla funzione `ungetc`, non sarà determinato finché tutti i caratteri rinviati non saranno letti o eliminati. Per uno stream binario, il suo indicatore di posizione del file sarà determinato da ogni chiamata successiva alla funzione `ungetc`; qualora prima di un'invocazione il suo valore sia zero, questo sarà indeterminato dopo la chiamata. La funzione `ungetc` restituirà il carattere rinviato dopo la conversione, o un `EOF` qualora l'operazione fallisca.

```
size_t fread(void *ptr, size_t dimensione, size_t nmemb, FILE *stream);
```

La funzione `fread` legge dallo stream puntato da `stream` un massimo di `nmemb` elementi di dimensione specificata da `dimensione`, immagazzinandoli nel vettore puntato da `ptr`. L'indicatore di posizione del file per lo stream (se definito) sarà fatto avanzare del numero di caratteri letti con successo. Qualora si verifichi un errore, il valore risultante dell'indicatore di posizione del file per lo stream sarà indeterminato. Qualora sia stato letto un elemento parziale, il suo valore sarà indeterminato.

La funzione `fread` restituirà il numero di elementi letti con successo, che potrà anche essere inferiore a `nmemb` qualora sia stato incontrato un errore di lettura o la fine del file. Nel caso che `dimensione` o `nmemb` sia zero, `fread` restituirà zero e i contenuti del vettore e lo stato dello stream resteranno immutati.

```
size_t fwrite(const void *ptr, size_t dimensione, size_t nmemb, FILE *stream);
```

La funzione `fwrite` scrive nello stream puntato da `stream` un massimo di `nmemb` elementi di dimensione specificata da `dimensione`, leggendoli dal vettore puntato da `ptr`. L'indicatore di posizione del file per lo stream (se definito) sarà fatto avanzare del numero di caratteri scritti con successo. Qualora si verifichi un errore, il valore risultante dell'indicatore di posizione del file per lo stream sarà indeterminato. La funzione `fwrite` restituirà il numero di elementi scritti con successo, che potrà essere inferiore a `nmemb` soltanto qualora si sia verificato un errore di scrittura.

```
int fgetpos(FILE *stream, fpos_t *pos);
```

La funzione `fgetpos` immagazzina nell'oggetto puntato da `pos` il valore corrente dell'indicatore di posizione relativo al file associato allo stream puntato da `stream`. Il valore immagazzinato conterrà delle informazioni non specificate, che potranno essere utilizzate dalla funzione `fsetpos` per riposizionare lo stream sulla posizione che aveva al momento della chiamata di `fgetpos`. In caso di esito positivo, la funzione `fgetpos` restituirà zero mentre, in caso di fallimento, restituirà un valore diverso da zero e immagazzinerà in `errno` un valore positivo definito dall'implementazione.

```
int fseek(FILE *stream, long int offset, int partenza);
```

La funzione `fseek` imposta l'indicatore di posizione del file per lo stream puntato da `stream`. Per uno stream binario, la nuova posizione, misurata in caratteri dall'inizio del file, sarà ottenuta aggiungendo `offset` alla posizione specificata da `partenza`. La posizione specificata corrisponderà all'inizio del file, qualora `partenza` sia `SEEK_SET`, o al valore corrente dell'indicatore di posizione del file, qualora sia `SEEK_CUR`, oppure alla fine dello stesso, qualora sia `SEEK_END`. Uno stream binario non ha bisogno di supportare in modo significativo le chiamate di `fseek` con un valore di `partenza` uguale a `SEEK_END`. Per uno stream di testo, `offset` dovrà essere zero, oppure un valore restituito da una precedente invocazione della funzione `ftell` sullo stesso stream e `partenza` dovrà essere `SEEK_SET`.

Una chiamata con esito positivo alla funzione `fseek` azzererà l'indicatore di fine del file per lo stream, e annullerà ogni effetto di `ungetc` sullo stesso. Dopo un'invocazione di `fseek`, l'operazione successiva su uno stream di aggiornamento potrà essere un input o un output. La funzione `fseek` restituirà un valore diverso da zero soltanto per una richiesta che non possa essere soddisfatta.

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

La funzione `fsetpos` imposta l'indicatore di posizione del file per lo stream puntato da `stream`, usando il valore dell'oggetto indicato da `pos`; tale valore dovrà essere stato ottenuto da una precedente invocazione della funzione `fgetpos` sullo stesso stream. Una chiamata con esito positivo alla funzione `fsetpos` azzererà l'indicatore di fine del file per lo stream, e annullerà ogni effetto di `ungetc` sullo stesso. Dopo un'invocazione di `fsetpos`, l'operazione successiva in uno stream di aggiornamento potrà essere un input o un output. Qualora abbia successo, la funzione `fsetpos` restituirà zero mentre, in caso di fallimento, restituirà un valore diverso da zero e immagazzinerà in `errno` un valore positivo definito dall'implementazione.

```
long int ftell(FILE *stream);
```

La funzione `ftell` ottiene il valore corrente dell'indicatore di posizione del file per lo stream puntato da `stream`. Per uno stream binario, il valore corrisponderà al numero di caratteri dall'inizio del file. Per uno stream di testo, il suo indicatore di posizione del file conterrà informazioni non specificate, che potranno essere utilizzate dalla funzione `fseek` per far ritornare l'indicatore di posizione del file per lo stream nel punto in cui si trovava al momento della chiamata a `ftell`; la differenza tra i due valori restituiti non è necessariamente una misura indicativa del numero di caratteri scritti o letti. In caso di successo, la funzione `ftell` restituirà il valore dell'indicatore di posizione del file per lo stream. In caso di fallimento, la funzione `ftell` restituirà `-1L` e immagazzinerà in `errno` un valore positivo definito dall'implementazione.

```
void rewind(FILE *stream);
```

La funzione `rewind` imposta l'indicatore di posizione del file per lo stream puntato da `stream` all'inizio del file. È equivalente a

```
(void) fseek(stream, 0L, SEEK_SET)
```

eccetto che sarà anche azzerato l'indicatore di errore per lo stream.

```
void clearerr(FILE *stream);
```

La funzione `clearerr` azzera gli indicatori di fine del file e di errore per lo stream puntato da `stream`.

```
int feof(FILE *stream);
```

La funzione `feof` controlla l'indicatore di fine del file per lo stream puntato da `stream`. La funzione `feof` restituirà un valore diverso da zero se e solo se sia stato impostato l'indicatore di fine del file per `stream`.

```
int ferror(FILE *stream);
```

La funzione `ferror` controlla l'indicatore di errore per lo stream puntato da `stream`. La funzione `ferror` restituirà un valore diverso da zero se e solo se sia stato impostato l'indicatore di errore per `stream`.

```
void perror(const char *s);
```

La funzione perror rileva il messaggio di errore corrispondente al numero presente nell'espressione intera errno. Essa scrive la seguente sequenza di caratteri nello stream dello standard error: in primo luogo (qualora il puntatore s e il carattere cui fa riferimento non siano nulli), la stringa puntata da s seguita dai due punti (:) e da uno spazio; quindi la stringa del messaggio di errore appropriata, seguita da un carattere newline. I contenuti delle stringhe del messaggio di errore saranno identici a quelli restituiti dalla funzione strerror con l'argomento errno e dipenderanno dall'implementazione.

G.11 Utilità generiche <stdlib.h>

EXIT_FAILURE
EXIT_SUCCESS

Espressioni intere che possono essere utilizzate come argomenti della funzione exit per restituire all'ambiente di esecuzione uno stato di chiusura con esito rispettivamente negativo o positivo.

MB_CUR_MAX

Un'espressione intera positiva, il cui valore corrisponde al numero massimo di byte contenuto in un carattere multibyte dell'insieme esteso, specificato dalla localizzazione corrente (categoria LC_CTYPE) e il cui valore non sarà mai maggiore di MB_LEN_MAX.

NULL

Una costante di tipo puntatore nullo definita dall'implementazione.

RAND_MAX

Un'espressione costante di intervallo, il cui valore corrisponde a quello massimo restituito dalla funzione rand. Il valore della macro RAND_MAX dovrà essere almeno 32767.

div_t

Un tipo di struttura corrispondente a quello del valore restituito dalla funzione div.

ldiv_t

Un tipo di struttura corrispondente a quello del valore restituito dalla funzione ldiv.

size_t

Il tipo intero senza segno restituito dall'operatore sizeof.

wchar_t

Un tipo intero, il cui intervallo di valori può rappresentare codici distinti per tutti i membri dell'insieme di caratteri più grande specificato tra quelli delle localizzazioni supportate. Il carattere nullo deve avere il valore di codice zero, mentre ogni membro dell'insieme di caratteri di base deve averne uno uguale a quello che si otterrebbe utilizzandolo da solo in una costante di carattere intera.

```
double atof(const char *nptr);
```

Converte la porzione iniziale della stringa puntata da nptr in una rappresentazione double. La funzione atof restituirà il valore convertito.

```
int atoi(const char *nptr);
```

Converte la porzione iniziale della stringa puntata da `nptr` in una rappresentazione `int`. La funzione `atoi` restituirà il valore convertito.

```
long int atol(const char *nptr);
```

Converte la porzione iniziale della stringa puntata da `nptr` in una rappresentazione `long`. La funzione `atol` restituirà il valore convertito.

```
double strtod(const char *nptr, char **endptr);
```

Converte la porzione iniziale della stringa puntata da `nptr` in una rappresentazione `double`. In primo luogo, essa scomporrà la stringa di input in tre parti: una sequenza iniziale, eventualmente vuota, formata da caratteri di spazio bianco (come specificato dalla funzione `isspace`), una sequenza soggetto somigliante a una costante in virgola mobile e una stringa finale formata da uno o più caratteri non riconosciuti, incluso quello nullo di terminazione della stringa di input. Quindi, essa tenterà di convertire la sequenza soggetto in un numero in virgola mobile e restituirà il risultato.

La forma espansa della sequenza soggetto sarà formata da: un segno positivo o negativo opzionale; una sequenza non vuota di cifre, contenente facoltativamente il carattere separatore dei decimali; una parte esponente opzionale, ma senza suffisso per valori in virgola mobile. La sequenza soggetto è definita come la sottosequenza più lunga della stringa di input, che incominci con il primo carattere diverso da quelli di spazio bianco e che sia della forma attesa. La sequenza soggetto non conterrà dei caratteri, qualora la stringa di input sia vuota, o sia formata interamente da spazi bianchi, o qualora il primo carattere diverso da uno spazio bianco non corrisponda a un segno, una cifra o al carattere separatore dei decimali.

Qualora la sequenza soggetto abbia la forma attesa, la sequenza di caratteri che incomincia con la prima cifra o con il carattere separatore dei decimali (ciò che appare prima) sarà interpretata come una costante in virgola mobile, eccetto che il carattere separatore dei decimali sarà utilizzato in sostituzione di un punto e che, qualora non compaia né un esponente né un carattere separatore dei decimali, si presumerà che questo segua l'ultima cifra della stringa. Qualora la sequenza soggetto cominci con un segno negativo, il valore risultante dalla conversione sarà invertito di segno. Un puntatore alla stringa finale sarà immagazzinato nell'oggetto puntato da `endptr`, a patto che questo non sia un puntatore nullo.

Qualora la sequenza soggetto sia vuota o non abbia la forma attesa, non sarà eseguita alcuna conversione; il valore di `nptr` sarà immagazzinato nell'oggetto puntato da `endptr`, a patto che questo non sia un puntatore nullo.

La funzione `strtod` restituirà il valore convertito, qualora ce ne sia uno. Qualora non possa essere eseguita nessuna conversione, sarà restituito uno zero. Qualora il valore corretto sia esterno all'intervallo di quelli rappresentabili, sarà restituito `HUGE_VAL` positivo o negativo (in accordo con il segno del valore), e sarà immagazzinato in `errno` il valore della macro `ERANGE`. Qualora il valore corretto sia troppo piccolo per essere rappresentato, sarà restituito zero e il valore della macro `ERANGE` sarà immagazzinato in `errno`.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Converte la porzione iniziale della stringa puntata da `nptr` in una rappresentazione `long int`. In primo luogo essa scomporrà la stringa di input in tre parti: una sequenza iniziale, eventualmente vuota, formata da caratteri di spazio bianco (così come specificato dalla funzione `isspace`), una sequenza soggetto somigliante a un intero rappresentato in una qualche base

determinata dal valore di base e una stringa finale formata da uno o più caratteri non riconosciuti, incluso quello nullo di terminazione della stringa di input. Quindi, essa tenterà di convertire la sequenza soggetto in un intero e restituirà il risultato.

Qualora il valore di base sia zero, la forma attesa della sequenza soggetto sarà quella di una costante intera, facoltativamente preceduta da un segno positivo o negativo, ma senza suffisso per gli interi. Qualora il valore di base sia compreso tra 2 e 36, la forma attesa sarà una sequenza di lettere e cifre che rappresenterà un intero nella base specificata da base, facoltativamente preceduta da un segno positivo o negativo, ma senza suffisso per gli interi. Le lettere da a (o A) a z (o Z) saranno associate ai valori da 10 a 35; saranno ammesse soltanto le lettere i cui valori ascritti siano inferiori a quello di base. Qualora il valore di base sia 16, i caratteri 0x o 0X potranno precedere facoltativamente la sequenza di lettere e cifre e seguire il segno, nel caso sia presente.

La sequenza soggetto è definita come la sottosequenza iniziale più lunga della stringa di input, che incominci con il primo carattere diverso da quelli di spazio bianco e che sia della forma attesa. La sequenza soggetto non conterrà dei caratteri, qualora la stringa di input sia vuota, o sia formata interamente da spazi bianchi, o qualora il primo carattere diverso da uno spazio bianco non corrisponda a un segno, a una lettera o a una cifra ammissibile.

Qualora la sequenza soggetto abbia la forma attesa e il valore di base sia zero, la sequenza di caratteri che incomincia con la prima cifra sarà interpretata come una costante intera. Qualora la sequenza soggetto abbia la forma attesa e base sia compresa tra 2 e 36, questa sarà utilizzata come base per la conversione, attribuendo a ogni lettera il suo valore così come detto in precedenza. Qualora la sequenza soggetto incominci con un segno negativo, il valore risultante dalla conversione sarà invertito di segno. Un puntatore alla stringa finale sarà immagazzinato nell'oggetto puntato da endptr, a patto che questo non sia un puntatore nullo.

Qualora la sequenza soggetto sia vuota o non abbia la forma attesa, non sarà eseguita alcuna conversione; il valore di nptr sarà immagazzinato nell'oggetto puntato da endptr, a patto che questo non sia un puntatore nullo.

La funzione strtol restituirà il valore convertito, qualora ce ne sia uno. Qualora non possa essere eseguita nessuna conversione, sarà restituito uno zero. Qualora il valore corretto sia esterno all'intervallo di quelli rappresentabili, sarà restituito LONG_MAX o LONG_MIN (in accordo con il segno del valore) e sarà immagazzinato in errno il valore della macro ERANGE.

```
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

Converte la porzione iniziale della stringa puntata da nptr in una rappresentazione unsigned long int. La funzione strtoul funziona in modo identico a strtol. La funzione strtoul restituirà il valore convertito, qualora ce ne sia uno. Qualora non possa essere eseguita nessuna conversione, sarà restituito uno zero. Qualora il valore corretto sia esterno all'intervallo di quelli rappresentabili, sarà restituito ULONG_MAX e il valore della macro ERANGE sarà immagazzinato in errno.

```
int rand(void);
```

La funzione rand calcola una sequenza di interi pseudocasuali compresi nell'intervallo da 0 a RAND_MAX. La funzione rand restituirà un intero pseudocasuale.

```
void srand(unsigned int seme);
```

Utilizza l'argomento come seme per una nuova sequenza di numeri pseudocasuali da restituire con le chiamate successive di `rand`. Qualora `srand` sia invocata successivamente con lo stesso valore di seme, la sequenza di numeri pseudocasuali sarà ripetuta. Qualora `rand` sia richiamata prima che sia stata effettuata una qualsiasi invocazione di `srand`, dovrà essere generata la stessa sequenza prodotta quando `srand` viene chiamata per la prima volta con un valore di seme uguale a 1. Le seguenti funzioni definiscono un'implementazione portabile di `rand` e `srand`.

```
static unsigned long int next = 1;

int rand(void)   /* si assume che RAND_MAX sia 32767 */
{
   next = next * 1103515245 +12345;
   return (unsigned int) (next/65536) % 32768;
}

void srand(unsigned int seed)
{
   next = seed;
}
```

```
void *calloc(size_t nmemb, size_t dimensione);
```

Alloca uno spazio per un vettore di `nmemb` oggetti, di dimensioni pari a `dimensione`. Lo spazio allocato sarà inizializzato con tutti i bit a zero. La funzione `calloc` restituirà un puntatore nullo o uno che faccia riferimento allo spazio allocato.

```
void free(void *ptr);
```

Fa in modo che lo spazio puntato da `ptr` sia rilasciato, ovverosia che sia reso disponibile per ulteriori allocazioni. Qualora `ptr` sia un puntatore nullo, non sarà eseguita nessuna azione. In caso contrario, qualora l'argomento non corrisponda a un puntatore restituito in precedenza dalle funzioni `calloc`, `malloc` o `realloc`, o qualora lo spazio sia stato rilasciato da una chiamata a `free` o a `realloc`, il comportamento sarà indefinito.

```
void *malloc(size_t dimensione);
```

Alloca uno spazio per un oggetto la cui dimensione sarà specificata da `dimensione` e il cui valore sarà indeterminato. La funzione `malloc` restituirà un puntatore nullo o uno che faccia riferimento allo spazio allocato.

```
void *realloc(void *ptr, size_t dimensione);
```

Cambia la dimensione dell'oggetto puntato da `ptr` con quella specificata da `dimensione`. I contenuti dell'oggetto non saranno modificati entro la dimensione più piccola tra la nuova e la vecchia. Qualora la nuova dimensione sia più grande, il valore della nuova porzione allocata per l'oggetto sarà indeterminato. Qualora `ptr` sia un puntatore nullo, la funzione `realloc` si comporterà come `malloc` per la dimensione specificata. Altrimenti, qualora `ptr` non corrisponda a un puntatore restituito in precedenza dalle funzioni `calloc`, `malloc` o `realloc`, o qualora lo spazio sia stato rilasciato da una chiamata alle funzioni `free` o `realloc`, il comportamento sarà indefinito. Qualora lo spazio non possa essere allocato, l'oggetto puntato da `ptr` resterà immutato. Qualora `dimensione` sia zero e `ptr` non sia nullo, l'oggetto puntato sarà rilasciato. La funzione `realloc` restituirà un puntatore nullo o uno che faccia riferimento allo spazio allocato ed eventualmente spostato.

```
void abort(void);
```

Fa in modo che si verifichi la chiusura anormale del programma, sempre che il segnale SIGABRT non sia intercettato e che il relativo gestore non restituisca il controllo. Lo svuotamento degli stream di output aperti, la chiusura degli stream aperti e la rimozione dei file temporanei, dipenderanno dall'implementazione. All'ambiente di esecuzione sarà restituita una forma definita dall'implementazione dello stato terminazione senza successo, attraverso la chiamata della funzione `raise(SIGABRT)`. La funzione `abort` non potrà restituire il controllo al chiamante.

```
int atexit(void (*funz)(void));
```

Registra la funzione puntata da `funz` perché sia richiamata senza argomenti alla terminazione normale del programma. L'implementazione dovrà supportare la registrazione di almeno 32 funzioni. La funzione `atexit` restituirà zero qualora la registrazione abbia successo, un valore diverso da zero qualora fallisca.

```
void exit(int stato);
```

Fa in modo che si verifichi la chiusura normale del programma. Qualora un programma esegua più di una chiamata alla funzione `exit`, il comportamento sarà indefinito. In primo luogo, saranno richiamate in ordine inverso alla loro registrazione tutte le funzioni registrate da `atexit`. Ognuna di esse sarà richiamata una volta per ogni registrazione effettuata. In seguito, saranno svuotati tutti gli stream aperti che abbiano nei buffer dei dati non ancora scritti, saranno chiusi tutti gli stream aperti e saranno rimossi tutti i file creati con la funzione `tmpfile`.

Infine, il controllo sarà restituito all'ambiente di esecuzione. Qualora il valore di `stato` sia zero o `EXIT_SUCCESS`, sarà restituita una forma definita dall'implementazione dello stato terminazione con successo. Qualora il valore di `stato` sia `EXIT_FAILURE`, sarà restituita una forma definita dall'implementazione dello stato terminazione senza successo. Altrimenti lo stato restituito sarà definito dall'implementazione. La funzione `exit` non potrà restituire il controllo al suo chiamante.

```
char *getenv(const char *nome);
```

Ricerca all'interno di un elenco di ambiente fornito da quello di esecuzione una stringa corrispondente a quella puntata da `nome`. L'insieme dei nomi dell'ambiente e dei metodi per alterare il relativo elenco saranno definiti dall'implementazione. Restituirà un puntatore a una stringa associata al membro corrispondente dell'elenco. La stringa puntata non dovrebbe essere modificata dal programma, ma potrà essere sostituita da una successiva chiamata alla funzione `getenv`. Qualora il `nome` specificato non possa essere ritrovato, sarà restituito un puntatore nullo.

```
int system(const char *stringa);
```

Passa la stringa puntata da `stringa` all'ambiente di esecuzione perché sia eseguita da un interprete di comandi in un modo definito dall'implementazione. Per `stringa` potrà essere utilizzato un puntatore nullo per verificare se esista un interprete di comandi. Qualora l'argomento sia un puntatore nullo, la funzione `system` restituirà un valore diverso da zero soltanto qualora sia disponibile un interprete di comandi. Qualora l'argomento non sia un puntatore nullo, la funzione `system` restituirà un valore definito dall'implementazione.

```
void *bsearch(const void *chiave, const void *base, size_t nmemb,
        size_t dimensione, int (*compar)(const void *, const void *));
```

Ricerca in un vettore di `nmemb` oggetti, il cui primo elemento sarà puntato da `base`, un oggetto che corrisponda a quello puntato da `chiave`. La dimensione di ogni elemento del vettore sarà specificata da `dimensione`. La funzione di comparazione puntata da `compar` sarà richiamata con due argomenti che punteranno rispettivamente all'oggetto `chiave` e a un elemento del vettore. La funzione dovrà restituire un intero minore, uguale o maggiore a zero qualora l'oggetto `chiave` sia considerato rispettivamente minore, uguale o maggiore di quello del vettore. Questo dovrà essere formato, nell'ordine, da tutti gli elementi considerati minori, uguali e maggiori dell'oggetto `chiave`.

La funzione `bsearch` restituirà un puntatore all'elemento corrispondente del vettore, o uno nullo qualora non sia stata ritrovata alcuna corrispondenza. Qualora siano considerati uguali due elementi, non sarà specificato quale dei due sarà quello corrispondente.

```
void qsort(void *base, size_t nmemb, size_t dimensione, int
      (*compar)(const void *, const void *));
```

Ordina un vettore di `nmemb` oggetti. L'elemento iniziale sarà puntato da `base`. La dimensione di ogni oggetto sarà specificata da `dimensione`. I contenuti del vettore saranno ordinati in modo ascendente secondo la funzione di comparazione puntata da `compar`; questa sarà richiamata con due argomenti che punteranno agli oggetti che dovranno essere confrontati. La funzione restituirà un intero minore, uguale o maggiore a zero qualora il primo argomento sia considerato rispettivamente minore, uguale o maggiore del secondo. Qualora due elementi siano considerati uguali, il loro ordine all'interno del vettore ordinato non sarà definito.

```
int abs(int j);
```

Calcola il valore assoluto di un intero `j`. Qualora il risultato non possa essere rappresentato, il comportamento sarà indefinito. La funzione `abs` restituirà il valore assoluto.

```
div_t div(int numer, int denom);
```

Calcola il quoziente e il resto della divisione del numeratore `numer` per il denominatore `denom`. Qualora la divisione non sia esatta, il quoziente risultante sarà l'intero più vicino per difetto al quoziente algebrico. Qualora il risultato non possa essere rappresentato, il comportamento sarà indefinito; altrimenti, `quoz * denom + res` dovrà essere uguale a `numer`. La funzione `div` restituirà una struttura di tipo `div_t`, comprendente il quoziente e il resto. La struttura dovrà contenere i seguenti membri, in qualsiasi ordine:

```
int quoz;    /* quoziente */
int res;     /* resto */
long int labs(long int j);
```

Simile alla funzione `abs`, eccetto che l'argomento e il valore restituito saranno di tipo `long int`.

```
ldiv_t ldiv(long int numer, long int denom);
```

Simile alla funzione `div`, eccetto che gli argomenti e i membri della struttura restituita (che sarà di tipo `ldiv_t`) saranno tutti dei `long int`.

```
int mblen(const char *s, size_t n);
```

Qualora s non sia un puntatore nullo, la funzione mblen determinerà il numero dei byte contenuti nel carattere multibyte puntato da s. Qualora s sia un puntatore nullo, la funzione mblen restituirà un valore diverso da zero o zero, a seconda che la codifica del carattere multibyte sia o no dipendente dallo stato. Qualora s non sia un puntatore nullo, la funzione mblen restituirà 0 (nel caso che s punti al carattere nullo), oppure il numero di byte del carattere multibyte (nel caso che i successivi n o meno caratteri formino un multibyte valido), oppure –1 (nel caso che i suddetti non formino un carattere multibyte valido).

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Qualora s non sia un puntatore nullo, la funzione mbtowc determinerà il numero dei byte contenuti nel carattere multibyte puntato da s. Essa quindi determinerà il codice per il valore di tipo wchar_t che corrisponda a quel carattere multibyte. (Il valore del codice corrispondente al carattere nullo è zero). Qualora il carattere multibyte sia valido e pwc non sia un puntatore nullo, la funzione mbtowc immagazzinerà il codice nell'oggetto puntato da pwc. Saranno esaminati al massimo n byte del vettore puntato da s.

Qualora s sia un puntatore nullo, la funzione mbtowc restituirà un valore diverso da zero o zero, a seconda che la codifica del carattere multibyte sia o no dipendente dallo stato. Qualora s non sia un puntatore nullo, la funzione mbtowc restituirà 0 (nel caso che s punti al carattere nullo), oppure il numero di byte contenuti nel carattere multibyte convertito (nel caso che i successivi n o meno byte formino un multibyte valido), oppure –1 (nel caso che i suddetti non formino un carattere multibyte valido). In nessun caso il valore restituito sarà maggiore di n o di quello della macro MB_CUR_MAX.

```
int wctomb(char *s, wchar_t wchar);
```

La funzione wctomb determina il numero di byte necessario per rappresentare il carattere multibyte corrispondente al codice il cui valore sarà contenuto in wchar (inclusa ogni modifica allo stato relativo al tasto delle maiuscole). Essa immagazzinerà la rappresentazione del carattere multibyte nell'oggetto di tipo vettore puntato da s (qualora s non sia un puntatore nullo). Saranno immagazzinati un massimo di MB_CUR_MAX caratteri. Qualora il valore di wchar sia zero, la funzione wctomb rimarrà nello stato iniziale relativo al tasto delle maiuscole.

Qualora s sia un puntatore nullo, la funzione wctomb restituirà un valore diverso da zero o zero, a seconda che la codifica del carattere multibyte sia o no dipendente dallo stato. Qualora s non sia un puntatore nullo, la funzione wctomb restituirà –1 qualora il valore di wchar non corrisponda a un carattere multibyte valido, oppure restituirà il numero di byte del carattere multibyte corrispondente al valore di wchar. In nessun caso il valore restituito sarà maggiore di quello della macro MB_CUR_MAX.

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

La funzione mbstowcs legge una sequenza di caratteri multibyte, che incominciano nello stato iniziale relativo al tasto delle maiuscole, dal vettore puntato da s e la converte nella sequenza di codici corrispondenti, immagazzinandone un massimo di n nel vettore puntato da pwcs. Non sarà esaminato o convertito nessun carattere multibyte successivo a quello nullo (che sarà convertito in un codice di valore zero). Ogni carattere multibyte sarà convertito come lo farebbe un'invocazione della funzione mbtowc, eccetto che lo stato relativo al tasto delle maiuscole nella funzione mbtowc non sarà influenzato.

Nel vettore puntato da pwcs saranno modificati al massimo n elementi. Il comportamento di una copia tra oggetti che si sovrappongano sarà indefinito. Qualora sia incontrato

un carattere multibyte non valido, la funzione `mbstowcs` restituirà `(size_t) - 1`. Altrimenti, la funzione `mbstowcs` restituirà il numero degli elementi modificati nel vettore, senza includere il codice zero di terminazione, qualora ce ne sia uno.

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

La funzione `wcstombs` legge dal vettore puntato da `pwcs` una sequenza di codici che corrispondono a dei caratteri multibyte, convertendola in una sequenza di caratteri multibyte che incominciano nello stato iniziale relativo al tasto delle maiuscole, e immagazzina i caratteri multibyte nel vettore puntato da `s`, fermandosi qualora un carattere multibyte faccia superare il limite di `n` byte, o qualora sia stato immagazzinato un carattere nullo. Ogni codice sarà convertito come lo farebbe un'invocazione della funzione `wctomb`, eccetto che in questo caso lo stato relativo al tasto delle maiuscole nella funzione `wctomb` non sarà influenzato.

Nel vettore puntato da `s` saranno modificati al massimo `n` byte. Qualora la copia avvenga tra oggetti che si sovrappongano, il risultato sarà indefinito. Qualora sia incontrato un codice che non corrisponda a un carattere multibyte valido, la funzione `wcstombs` restituirà `(size_t) - 1`. Altrimenti, la funzione `wcstombs` restituirà il numero dei byte modificati, senza includere il carattere nullo di terminazione, qualora ce ne sia uno.

G.12 Gestione delle stringhe <string.h>

```
NULL
```

Una costante di tipo puntatore nullo definita dall'implementazione.

```
size_t
```

Il tipo intero senza segno restituito dall'operatore `sizeof`.

```
void *memcpy(void *s1, const void *s2, size_t n);
```

La funzione `memcpy` copia `n` caratteri dall'oggetto puntato da `s2` in quello puntato da `s1`. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione `memcpy` restituirà il valore di `s1`.

```
void *memmove(void *s1, const void *s2, size_t n);
```

La funzione `memmove` copia `n` caratteri dall'oggetto puntato da `s2` in quello puntato da `s1`. La copia avverrà come se i caratteri dell'oggetto puntato da `s2` fossero prima copiati in un vettore temporaneo di `n` caratteri, che non si sovrapponga agli oggetti puntati da `s1` e `s2`, e quindi fossero ricopiati dal vettore temporaneo nell'oggetto puntato da `s1`. La funzione `memmove` restituirà il valore di `s1`.

```
char *strcpy(char *s1, const char *s2);
```

La funzione `strcpy` copia la stringa puntata da `s2` (incluso il carattere nullo di terminazione) nel vettore puntato da `s1`. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione `strcpy` restituirà il valore di `s1`.

```
char *strncpy(char *s1, const char *s2, size_t n);
```

La funzione `strncpy` copia un massimo di `n` caratteri, escludendo quelli successivi a quello nullo, dal vettore puntato da `s2` in quello puntato da `s1`. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. Qualora il vettore puntato

da s2 sia una stringa più corta di n caratteri, nella copia del vettore puntato da s1 saranno accodati dei caratteri nulli, finché non sarà stato raggiunto il numero indicato da n. La funzione strncpy restituirà il valore di s1.

```
char *strcat(char *s1, const char *s2);
```

La funzione strcat accoda una copia della stringa puntata da s2 (incluso il carattere nullo di terminazione) alla fine di quella puntata da s1. Il carattere iniziale di s2 si sostituirà a quello di terminazione di s1. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione strcat restituirà il valore di s1.

```
char *strncat(char *s1, const char *s2, size_t n);
```

La funzione strncat accoda un massimo di n caratteri (quello nullo e quelli che lo seguono non saranno accodati) dal vettore puntato da s2 alla fine della stringa indicata da s1. Il carattere iniziale di s2 si sostituirà a quello nullo di terminazione di s1. Al risultato sarà sempre accodato un carattere nullo di terminazione. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione strncat restituirà il valore di s1.

```
int memcmp(const void *s1, const void *s2, size_t n);
```

La funzione memcmp confronta i primi n caratteri dell'oggetto puntato da s1 con i primi n caratteri di quello puntato da s2. La funzione memcmp restituirà un intero maggiore, uguale o minore di zero qualora l'oggetto puntato da s1 sia maggiore, uguale o minore di quello puntato da s2.

```
int strcmp(const char *s1, const char *s2);
```

La funzione strcmp confronta la stringa puntata da s1 con quella puntata da s2. La funzione strcmp restituirà un intero maggiore, uguale o minore di zero qualora l'oggetto puntato da s1 sia maggiore, uguale o minore di quello puntato da s2.

```
int strcoll(const char *s1, const char *s2);
```

La funzione strcoll confronta la stringa puntata da s1 con quella puntata da s2, interpretando entrambe in modo conforme a quanto indicato dalla categoria LC_COLLATE della localizzazione corrente. La funzione strcoll restituirà un intero maggiore, uguale o minore di zero qualora la stringa puntata da s1 sia maggiore, uguale o minore di quella puntata da s2, quando entrambe siano interpretate in modo conforme alla localizzazione corrente.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

La funzione strncmp confronta un massimo di n caratteri del vettore puntato da s1, escludendo quelli successivi a quello nullo, con i caratteri corrispondenti del vettore puntato da s2. La funzione strncmp restituirà un valore maggiore, uguale o minore di zero qualora il vettore puntato da s1, eventualmente terminato dal carattere nullo, sia maggiore, uguale o minore di quello puntato da s2, eventualmente terminato dal carattere nullo.

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

La funzione strxfrm trasforma la stringa puntata da s2, inserendo quella risultante nel vettore indicato da s1. La trasformazione sarà tale che, qualora la funzione strcmp fosse applicata a due stringhe trasformate, essa restituirebbe un valore maggiore, uguale o minore di zero corrispondente al risultato della funzione strcoll applicata alle stesse due stringhe

originali. Nel vettore risultante puntato da s1 saranno inseriti al massimo n caratteri, incluso quello nullo di terminazione. Qualora n sia zero, s1 potrà essere un puntatore nullo. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. La funzione strxfrm restituirà la lunghezza della stringa trasformata (escluso il carattere nullo di terminazione). Qualora il valore restituito sia pari o maggiore di n, i contenuti del vettore puntato da s1 saranno indeterminati.

```
void *memchr(const void *s, int c, size_t n);
```

La funzione memchr individua la prima occorrenza di c (convertito in un unsigned char) nei primi n caratteri (ognuno interpretato come unsigned char) dell'oggetto puntato da s. La funzione memchr restituirà un puntatore al carattere individuato, o uno nullo qualora il carattere non sia presente nell'oggetto.

```
char *strchr(const char *s, int c);
```

La funzione strchr individua la prima occorrenza di c (convertito in un char) nella stringa puntata da s. Il carattere nullo di terminazione sarà considerato parte della stringa. La funzione strchr restituirà un puntatore al carattere individuato, o uno nullo qualora il carattere non sia presente nella stringa.

```
size_t strcspn(const char *s1, const char *s2);
```

La funzione strcspn calcola la lunghezza del segmento iniziale massimo che, nella stringa puntata da s1, sia formato interamente da caratteri non contenuti nella stringa puntata da s2. La funzione strcspn restituirà la lunghezza del segmento.

```
char *strpbrk(const char *s1, const char *s2);
```

La funzione strpbrk individua la prima occorrenza nella stringa puntata da s1 di qualsiasi carattere incluso in quella puntata da s2. La funzione strpbrk restituirà un puntatore al carattere, o uno nullo qualora nessuno di quelli inclusi in s2 sia presente in s1.

```
char *strrchr(const char *s, int c);
```

La funzione strrchr individua l'ultima occorrenza di c (convertito in un char) nella stringa puntata da s. Il carattere nullo di terminazione sarà considerato parte della stringa. La funzione strrchr restituirà un puntatore al carattere, o uno nullo qualora c non sia presente nella stringa.

```
size_t strspn(const char *s1, const char *s2);
```

La funzione strspn calcola la lunghezza del segmento iniziale massimo che, nella stringa puntata da s1, sia formato interamente da caratteri contenuti nella stringa puntata da s2. La funzione strspn restituirà la lunghezza del segmento.

```
char *strstr(const char *s1, const char *s2);
```

La funzione strstr individua, nella stringa puntata da s1, la prima occorrenza della sequenza di caratteri (escluso quello nullo di terminazione) inclusa nella stringa puntata da s2. La funzione strstr restituirà un puntatore alla stringa individuata, o uno nullo qualora non sia stata ritrovata nessuna corrispondenza. Qualora s2 punti a una stringa di lunghezza zero, la funzione restituirà s1.

```
char *strtok(char *s1, const char *s2);
```

Una serie di invocazioni della funzione `strtok` suddividerà la stringa puntata da `s1` in una sequenza di token, ognuno dei quali sarà delimitato da un carattere tra quelli inclusi nella stringa puntata da `s2`. La prima invocazione della serie avrà `s1` come suo argomento e sarà seguita da invocazioni che abbiano un puntatore nullo come loro primo argomento. La stringa dei separatori puntata da `s2` potrà cambiare da una chiamata all'altra.

La prima invocazione della serie ricercherà, nella stringa puntata da `s1`, il primo carattere che non sia contenuto in quella dei separatori puntata da `s2`. Qualora il suddetto carattere non sia ritrovato, allora non ci saranno token nella stringa puntata da `s1` e la funzione `strtok` restituirà un puntatore nullo. Qualora il suddetto carattere sia stato ritrovato, questo corrisponderà all'inizio del primo token.

La funzione `strtok` ricercherà quindi da quel punto un carattere che sia tra quelli contenuti nella stringa dei separatori. Qualora il suddetto carattere non sia stato ritrovato, il token corrente si estenderà fino alla fine della stringa puntata da `s1` e le successive ricerche di un token restituiranno un puntatore nullo. Qualora il suddetto carattere sia stato ritrovato, questo sarà sostituito da uno nullo che terminerà il token corrente. La funzione `strtok` salverà un puntatore al carattere successivo, dal quale partirà la prossima ricerca di un token.

Ogni chiamata successiva, che abbia un puntatore nullo come valore del primo argomento, inizierà la ricerca dal punto salvato e si comporterà come descritto prima. L'implementazione dovrà comportarsi come se nessuna funzione della libreria richiami `strtok`. La funzione `strtok` restituirà un puntatore al primo carattere di un token, o uno nullo qualora non ce ne siano.

```
void *memset(void *s, int c, size_t n);
```

La funzione `memset` copia il valore di `c`, convertito in un `unsigned char`, in ognuno dei primi `n` caratteri dell'oggetto puntato da `s`. La funzione `memset` restituirà il valore di `s`.

```
char *strerror(int errnum);
```

La funzione `strerror` individua la stringa del messaggio di errore corrispondente al valore di `errnum`. L'implementazione dovrà comportarsi come se nessuna funzione della libreria richiami `strerror`. La funzione `strerror` restituirà un puntatore alla stringa, i cui contenuti saranno definiti dall'implementazione. Il vettore puntato non dovrebbe essere modificato dal programma, ma potrà essere sostituito da una chiamata successiva alla funzione `strerror`.

```
size_t strlen(const char *s);
```

La funzione `strlen` calcola la lunghezza della stringa puntata da `s`. La funzione `strlen` restituirà il numero di caratteri che precedono quello nullo di terminazione.

G.13 Data e ora <time.h>

`CLOCKS_PER_SEC`

Il numero per secondo del valore restituito dalla funzione `clock`.

`NULL`

Una costante di tipo puntatore nullo definita dall'implementazione.

`clock_t`

Un tipo aritmetico in grado di rappresentare l'ora.

```
time_t
```

Un tipo aritmetico in grado di rappresentare l'ora.

```
size_t
```

Il tipo intero senza segno restituito dell'operatore `sizeof`.

```
struct tm
```

Mantiene i componenti delle date, chiamati tempo frammentato. La struttura dovrà contenere almeno i seguenti membri, in qualsiasi ordine. Le semantiche dei membri e dei loro normali intervalli sono espresse nei commenti.

```
int tm_sec;     /* secondi dopo il minuto: [0, 59] */
int tm_min;     /* minuti dopo l'ora: [0, 59] */
int tm_hour;        /* ore dalla mezzanotte: [0, 23] */
int tm_mday;        /* giorno del mese: [1, 31] */
int tm_mon;     /* mese da gennaio: [0, 11] */
int tm_year;        /* anno dal 1900 */
int tm_wday;        /* giorni dalla domenica: [0, 6] */
int tm_yday;        /* giorni dal 1° gennaio: [0, 365] */
int tm_isdst;       /* flag per l'ora legale */
```

Il valore di `tm_isdst` sarà positivo qualora l'ora legale sia in corso, zero qualora non lo sia e negativo qualora l'informazione non sia disponibile.

```
clock_t clock(void);
```

La funzione `clock` determina il tempo di processore utilizzato. La funzione `clock` restituirà la miglior approssimazione dell'implementazione per quanto riguarda il tempo del processore usato dal programma, dall'inizio di un momento definito dall'implementazione e correlato soltanto con l'invocazione del programma. Per determinare il tempo in secondi, il valore restituito dalla funzione `clock` dovrà essere diviso per quello della macro `CLOCKS_PER_SEC`. Qualora il tempo usato del processore non sia disponibile o il suo valore non possa essere rappresentato, la funzione restituirà il valore `(clock_t) - 1`.

```
double difftime(time_t time1, time_t time0);
```

La funzione `difftime` calcola la differenza tra due date: `time1 - time0`. La funzione `difftime` restituirà la differenza espressa in secondi in un `double`.

```
time_t mktime(struct tm *timeptr);
```

La funzione `mktime` converte il tempo frammentato, espresso come ora locale, della struttura puntata da `timeptr` in un valore di tipo data con la stessa codifica di quelli restituiti dalla funzione `time`. I valori originali dei membri `tm_wday` e `tm_yday` della struttura saranno ignorati, mentre quelli degli altri membri non saranno limitati agli intervalli indicati prima. In un completamento successivo, i valori dei membri `tm_wday` e `tm_yday` della struttura saranno impostati in modo appropriato, mentre gli altri membri saranno impostati in modo da rappresentare la data specificata, ma con i loro valori forzati negli intervalli indicati prima; il valore finale di `tm_mday` non sarà impostato finché `tm_mon` e `tm_year` non saranno stati determinati. La funzione `mktime` restituirà la data specificata codificata come un valore di

tipo time_t. Qualora la data non possa essere rappresentata, la funzione restituirà il valore (time_t) - 1.

```
time_t time(time_t *timer);
```

La funzione time determina la data e l'ora corrente. Essa restituisce la miglior approssimazione dell'implementazione per quanto riguarda la data e l'ora correnti. Il valore (time_t) - 1 sarà restituito qualora la data e l'ora non siano disponibili. Qualora timer non sia un puntatore nullo, il valore restituito sarà assegnato anche all'oggetto puntato dallo stesso.

```
char *asctime(const struct tm *timeptr);
```

La funzione asctime converte il tempo frammentato della struttura puntata da timeptr in una stringa dal seguente formato

```
Sun Sep 16 01:03:52 1973\n\0
```

La funzione asctime restituirà un puntatore alla stringa.

```
char *ctime(const time_t *timer);
```

La funzione ctime converte la data e l'ora puntate da timer in una stringa che rappresenti l'ora locale. È equivalente a

```
asctime(localtime(timer))
```

La funzione ctime restituirà il puntatore ricevuto dalla chiamata della funzione asctime con quell'argomento di tempo frammentato.

```
struct tm *gmtime(const time_t *timer);
```

La funzione gmtime converte la data e l'ora puntate da timer in un tempo frammentato, espresso come Coordinated Universal Time (UTC, ora mondiale coordinata). La funzione gmtime restituirà un puntatore a quell'oggetto, o uno nullo qualora l'UTC non sia disponibile.

```
struct tm *localtime(const time_t *timer);
```

La funzione localtime converte la data e l'ora puntate da timer in un tempo frammentato espresso come ora locale. La funzione localtime restituirà un puntatore a quell'oggetto.

```
size_t strftime(char *s, size_t maxdim, const char *formato, const
          struct tm *timeptr);
```

La funzione strftime inserisce dei caratteri nel vettore puntato da s nel modo indicato dalla stringa puntata da formato. Questa sarà formata da zero o più specifiche di conversione e da caratteri multibyte ordinari. Tutti i caratteri ordinari (incluso quello nullo di terminazione) saranno copiati senza modifiche nel vettore. Qualora la copia avvenga tra oggetti che si sovrappongano, il comportamento sarà indefinito. Nel vettore saranno inseriti al massimo maxdim caratteri. Ogni specifica di conversione sarà sostituita dai caratteri appropriati come descritto nella lista seguente. I caratteri appropriati saranno determinati dalla categoria LC_TIME della localizzazione corrente e dai valori contenuti nella struttura puntata da timeptr.

%a sarà sostituita dal nome locale abbreviato del giorno della settimana.
%A sarà sostituita dal nome locale completo del giorno della settimana.
%b sarà sostituita dal nome locale abbreviato del mese.
%B sarà sostituita dal nome locale completo del mese.

`%c` sarà sostituita dall'appropriata rappresentazione locale della data e dell'ora.

`%d` sarà sostituita dal giorno del mese espresso come numero decimale (`01-31`).

`%H` sarà sostituita dall'ora (nel formato di 24 ore) espressa come numero decimale (`00-23`).

`%I` sarà sostituita dall'ora (nel formato di 12 ore) espressa come numero decimale (`01-12`).

`%j` sarà sostituita dal giorno dell'anno espresso come numero decimale (`001-366`).

`%m` sarà sostituita dal mese espresso come numero decimale (`01-12`).

`%M` sarà sostituita dal minuto espresso come numero decimale (`00-59`).

`%p` sarà sostituita dall'equivalente locale della designazione AM/PM associata ad un formato di 12 ore.

`%S` sarà sostituita dal secondo espresso come numero decimale (`00-59`).

`%U` sarà sostituita dal numero della settimana dell'anno (la prima domenica sarà il primo giorno della settimana 1) espresso come numero decimale (`00-53`).

`%w` sarà sostituita dal giorno della settimana espresso come numero decimale (`0-6`), dove la domenica sarà `0`.

`%W` sarà sostituita dal numero della settimana dell'anno (il primo lunedì sarà il primo giorno della settimana 1) espresso come numero decimale (`00-53`).

`%x` sarà sostituita dall'appropriata rappresentazione locale della data.

`%X` sarà sostituita dall'appropriata rappresentazione locale dell'ora.

`%y` sarà sostituita dall'anno senza il secolo espresso come numero decimale (`00-99`).

`%Y` sarà sostituita dall'anno (compreso il secolo) espresso come numero decimale.

`%Z` sarà sostituita dal nome o dall'abbreviazione della zona del fuso orario, o da nessun carattere qualora la zona del fuso orario non sia determinabile.

`%%` sarà sostituita da %.

Nel caso che una specifica di conversione non sia una di quelle elencate sopra, il comportamento sarà indefinito. Qualora il numero totale di caratteri risultanti, incluso quello nullo di terminazione, non sia maggiore di `maxdim`, la funzione `strftime` restituirà il numero di quelli inseriti nel vettore puntato da `s`, senza includere quello nullo di terminazione. Altrimenti, sarà restituito il valore zero e i contenuti del vettore saranno indeterminati.

G.14 Limiti dell'implementazione

<limits.h>

Le seguenti costanti simboliche dovranno essere definite con grandezza (valore assoluto) uguale o maggiore ai valori indicati di seguito.

```
#define CHAR_BIT                    8
```

Il numero di bit per l'oggetto più piccolo che non sia un campo di bit (byte).

```
#define SCHAR_MIN                 -127
```

Il valore minimo per un oggetto di tipo `signed char`.

```
#define SCHAR_MAX                 +127
```

Il valore massimo per un oggetto di tipo `signed char`.

`#define UCHAR_MAX` 255

Il valore massimo per un oggetto di tipo `unsigned char`.

`#define CHAR_MIN` `0` *o* `SCHAR_MIN`

Il valore minimo per un oggetto di tipo `char`.

`#define CHAR_MAX` `UCHAR_MAX` *o* `SCHAR_MAX`

Il valore massimo per un oggetto di tipo `char`.

`#define MB_LEN_MAX` 1

Il numero massimo di byte in un carattere multibyte, per ogni localizzazione supportata.

`#define SHRT_MIN` -32767

Il valore minimo per un oggetto di tipo `short int`.

`#define SHRT_MAX` +32767

Il valore massimo per un oggetto di tipo `short int`.

`#define USHRT_MAX` 65535

Il valore massimo per un oggetto di tipo `unsigned short int`.

`#define INT_MIN` -32767

Il valore minimo per un oggetto di tipo `int`.

`#define INT_MAX` +32767

Il valore massimo per un oggetto di tipo `int`.

`#define UINT_MAX` 65535

Il valore massimo per un oggetto di tipo `unsigned int`.

`#define LONG_MIN` -2147483647

Il valore minimo per un oggetto di tipo `long int`.

`#define LONG_MAX` +2147483647

Il valore massimo per un oggetto di tipo `long int`.

`#define ULONG_MAX` 4294967295

Il valore massimo per un oggetto di tipo `unsigned long int`.

<float.h>

`#define FLT_ROUNDS`

La modalità di arrotondamento per l'addizione in virgola mobile.

`-1` indeterminabile
`0` prossima allo zero
`1` al più vicino

2 prossima all'infinito positivo
3 prossima all'infinito negativo

Le seguenti costanti simboliche dovranno essere definite con grandezza (valore assoluto) uguale o maggiore dei valori indicati di seguito.

```
#define FLT_RADIX                    2
```

La base della rappresentazione dell'esponente, b.

```
#define FLT_MANT_DIG
#define LDBL_MANT_DIG
#define DBL_MANT_DIG
```

Il numero di cifre in base FLT_RADIX nel significando in virgola mobile, p.

```
#define FLT_DIG                      6
#define DBL_DIG                      10
#define LDBL_DIG                     10
```

Il numero di cifre decimali, q, tale che ogni numero in virgola mobile con q cifre decimali possa essere arrotondato in un numero in virgola mobile con base p e b cifre e riportato al suo valore originario senza modifiche alle q cifre decimali.

```
#define FLT_MIN_EXP
#define DBL_MIN_EXP
#define LDBL_MIN_EXP
```

L'intero negativo minimo tale che FLT_RADIX elevato a quella potenza meno 1 sia un numero in virgola mobile normalizzato.

```
#define FLT_MIN_10_EXP               -37
#define DBL_MIN_10_EXP               -37
#define LDBL_MIN_10_EXP              -37
```

L'intero negativo minimo tale che 10 elevato a quella potenza sia compreso nell'intervallo dei numeri in virgola mobile normalizzati.

```
#define FLT_MAX_EXP
#define DBL_MAX_EXP
#define LDBL_MAX_EXP
```

L'intero massimo tale che FLT_RADIX elevato a quella potenza meno 1 sia un numero finito rappresentabile in virgola mobile.

```
#define FLT_MAX_10_EXP               +37
#define DBL_MAX_10_EXP               +37
#define LDBL_MAX_10_EXP              +37
```

L'intero massimo tale che 10 elevato a quella potenza sia compreso nell'intervallo dei numeri finiti rappresentabili in virgola mobile.

Le seguenti costanti simboliche dovranno essere definite uguali o maggiori dei valori mostrati di seguito.

```
#define FLT_MAX                      1E+37
#define DBL_MAX                      1E+37
#define LDBL_MAX                     1E+37
```

Il numero massimo finito in virgola mobile rappresentabile.

Le seguenti costanti simboliche dovranno essere definite uguali o minori dei valori mostrati di seguito.

```
#define FLT_EPSILON                1E-5
#define DBL_EPSILON                1E-9
#define LDBL_EPSILON               1E-9
```

La differenza tra il valore 1,0 e quello minimo maggiore di 1,0 che sia rappresentabile nel tipo in virgola mobile specificato.

```
#define FLT_MIN                    1E-37
#define DBL_MIN                    1E-37
#define LDBL_MIN                   1E-37
```

Il numero positivo minimo normalizzato in virgola mobile.

(Diritti d'autore: questo materiale è stato condensato e adattato a partire dal documento American National Standard for Information Systems – Programming Language – C, ANSI/ISO 9899: 1990. Copie di questo standard possono essere acquistate dalla American National Standards Institute, West 42nd Street, New York, NY 10036.)

Appendice H

Programmare giochi: risolvere il Sudoku

H.1 Introduzione

Il gioco del Sudoku ha visto la propria popolarità esplodere a livello mondiale nel 2005. Quasi tutti i principali giornali ormai pubblicano un Sudoku al giorno. Videogame portatili consentono di giocare in qualsiasi momento, in ogni luogo e di creare, su richiesta, delle sfide di questo tipo a vari livelli di difficoltà.

Un *Sudoku completo* è una griglia 9×9 (ovvero, una matrice) in cui delle cifre da 1 a 9 compaiono una ed una sola volta in ogni riga, colonna e griglia 3×3. Nella griglia 9×9 parzialmente completa in Figura H.1, la riga 1, la colonna 1 e la griglia 3×3 nell'angolo in alto a sinistra della scacchiera contengono tutte delle cifre da 1 a 9 una ed una volta soltanto. Si noti che vengono utilizzate le convenzioni bidimensionali del C sulla numerazione delle righe e delle colonne, ma vengono ignorate la riga 0 e la colonna 0 in accordo alle convenzioni vigenti nella comunità degli appassionati del Sudoku.

Il tipico gioco del Sudoku fornisce molte caselle piene e molte caselle vuote, spesso organizzate secondo uno schema simmetrico, come è tipico nelle parole crociate. Il compito del giocatore è quello di riempire le caselle vuote per completare il gioco. Alcuni di questi sono facili da risolvere; altri sono abbastanza difficili, richiedendo delle strategie di soluzione sofisticate.

	1	2	3	4	5	6	7	8	9
1	5	1	3	4	9	7	6	2	8
2	4	6	8						
3	7	9	2						
4	2								
5	9								
6	3								
7	8								
8	1								
9	6								

Figura H.1 Griglia 9×9 di un Sudoku parzialmente completato

Verranno discusse varie semplici strategie di soluzione e verrà suggerito cosa fare quando queste falliscono. Verranno anche presentati vari approcci alla programmazione di creatori e risolvitori di giochi del Sudoku in C. Sfortunatamente lo Standard C non include primitive per la grafica e le GUI (*graphical user interface*), quindi la presentazione della scacchiera non sarà così elegante come potrebbe essere in Java ed altri linguaggi di programmazione che supportano tali caratteristiche.

H.2 Il Sudoku Resource Center della Deitel

È possibile consultare il *Sudoku Resource Center* della Deitel all'indirizzo `www.deitel.com/sudoku`: contiene download, tutorial, libri, e-book ed altro materiale che può essere di aiuto per padroneggiare il gioco. Si può seguire la storia del Sudoku dalle sue origini nell'ottavo secolo fino ai tempi moderni. È possibile scaricare diversi puzzle Sudoku a vari livelli di difficoltà, partecipare a gare per vincere libri sul Sudoku e ottenere un puzzle Sudoku giornaliero da inserire nel proprio sito web. Sono disponibili risorse importanti per i principianti, tra le quali le regole del gioco, consigli sulla risoluzione dei giochi più semplici, le migliori strategie di risoluzione e risolutori di Sudoku gratuiti: è sufficiente digitare il gioco preso dal proprio giornale o dal sito di Sudoku preferito ed ottenere una soluzione immediata; alcuni risolutori forniscono addirittura spiegazioni dettagliate passo per passo. È possibile scaricare ed installare Sudoku in versione mobile per telefoni cellulari, palmari, Game Boy© e dispositivi che supportano Java. Alcuni siti dedicati al Sudoku, oltre a suggerimenti sul gioco, forniscono dei timer e dei segnali che indicano quando un numero scorretto viene inserito. È possibile acquistare T-shirt e tazze con Sudoku stampati su di esse, partecipare ai forum, ottenere dei fogli di lavoro vuoti che possono essere stampati e provare dispositivi portatili: uno di questi offre un milione di varianti di gioco e cinque livelli di difficoltà. Si può scaricare del software per la creazione di nuovi giochi. E non soltanto per i deboli di cuore: si possono provare Sudoku diabolicamente difficili con trucchi contorti, un Sudoku circolare ed una variante del gioco con cinque griglie interconnesse. È possibile infine abbonarsi alla newsletter gratuita *The Deitel Buzz Online* all'indirizzo `www.deitel.com`, per ricevere notifiche di aggiornamenti al Sudoku Resource Center e ad altri Resource Center della Deitel che forniscono giochi, enigmi ed altri interessanti progetti di programmazione.

H.3 Strategie di soluzione

Quando si farà riferimento ad una griglia 9x9 di Sudoku, si userà il nome di matrice s. Guardando tutte le caselle piene nella riga, colonna e griglia 3x3 che include una determinata casella vuota, il valore di quest'ultima può diventare ovvio. Banalmente il valore della casella s[1][7] in Figura H.2 deve essere 6.

Meno banalmente, per determinare il valore di s[1][7] in Figura H.3, bisogna cogliere dei suggerimenti dalla riga 1 (ovvero, bisogna considerare le cifre 3, 6 e 9) e dalla griglia 3×3 in alto a destra (ovvero, bisogna considerare le cifre 9, 8, 4 e 2). In questo caso la casella vuota s[1][7] deve assumere il valore 5: l'unico numero non ancora citato in riga 1, colonna 7 e nella griglia 3×3 in alto a destra.

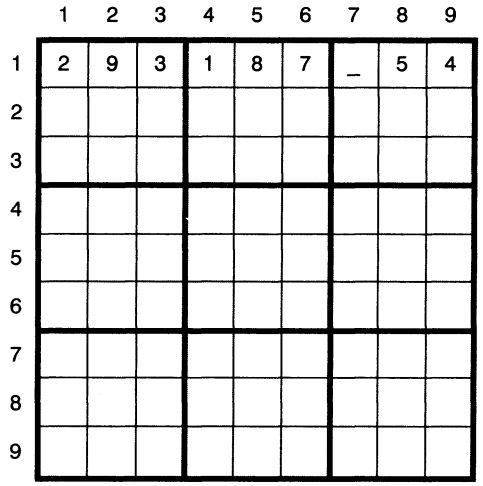


Figura H.2 Determinare il valore di una casella controllando tutte le caselle piene nella stessa riga

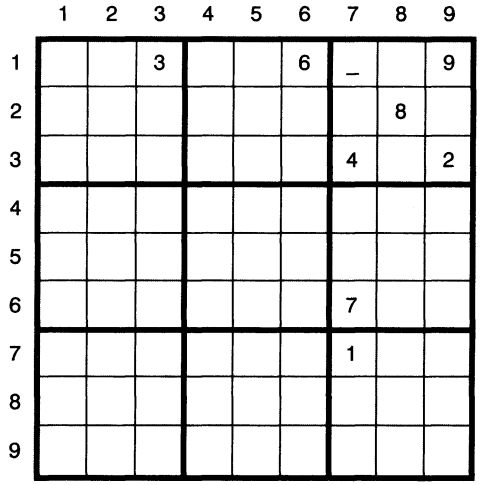


Figura H.3 Determinare il valore di una casella controllando tutte le caselle piene nella stessa riga, colonna e griglia 3×3

Singoletti

Le strategie che sono state discusse fino ad ora possono determinare facilmente le cifre finali per qualche casella aperta, ma spesso è necessario investigare più profondamente. La colonna 6 in Figura H.4 mostra delle caselle i cui valori sono già determinati (ovvero, s[1][6] contiene 9, s[3][6] contiene 6 ecc.) e delle caselle che indicano un insieme di valori (che chiameremo "possibilità") che al momento corrente sono ancora plausibili per tali caselle.

La casella s[6][6] contiene 257, indicando che soltanto i valori 2, 5 o 7 possono eventualmente esserle assegnati. Le altre due caselle aperte nella colonna 6, s[2][6] e s[5][6], conten-

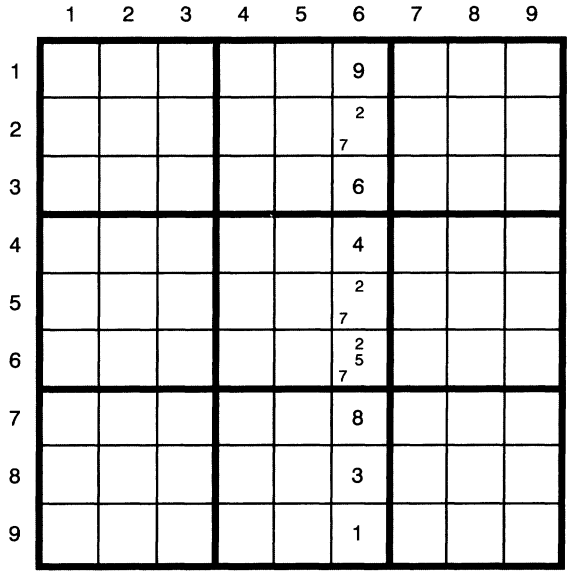


Figura H.4 Notazione che illustra gli insiemi completi dei possibili valori per le caselle aperte

gono entrambe 27, indicando che soltanto i valori 2 o 7 possono eventualmente essere loro assegnati. Quindi s[6][6], la sola casella nella colonna 6 che elenca 5 come un possibile valore rimanente, deve assumere il valore 5, che quindi viene definito un *singoletto*. Così si può fissare il valore della casella s[6][6] a 5 (Figura H.5), semplificando in qualche modo il problema.

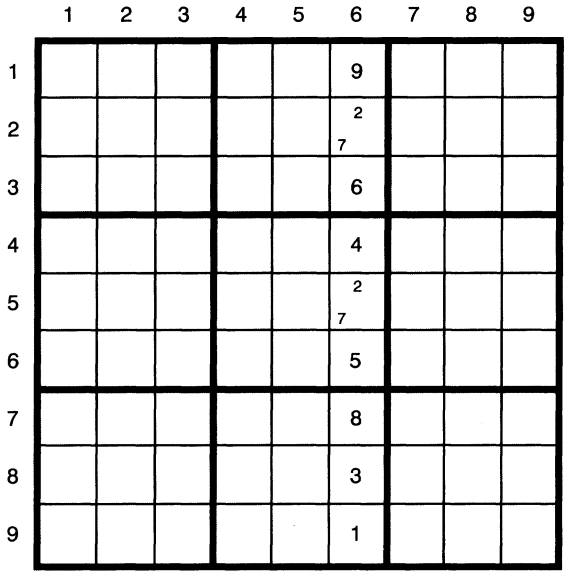


Figura H.5 Fissare il valore della casella s[6][6] al singoletto 5

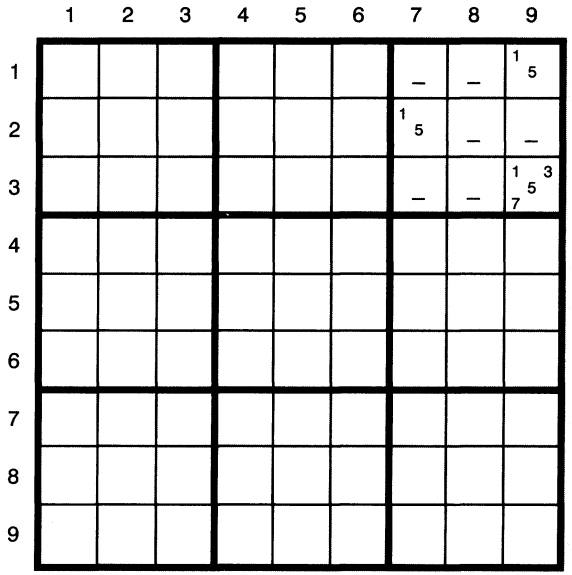


Figura H.6 Usare le doppie per semplificare il puzzle

Doppie

Si consideri la griglia 3×3 in alto a destra in Figura H.6. Le caselle barrate potrebbero essere già fissate o potrebbero avere una lista di valori possibili. Si noti la presenza di *doppie*: le due caselle s[1][9] e s[2][7] contengono soltanto le due possibilità 15. Se s[1][9] alla fine assume il valore 1, allora s[2][7] deve assumere il valore 5; se s[1][9] alla fine assume il valore 5, allora s[2][7] deve assumere il valore 1. Quindi fra loro tali caselle utilizzeranno alla fine il valore 1 ed il valore 5. In tal modo 1 e 5 possono essere eliminati dalla casella s[3][9] che contiene i valori possibili 1357: così si può riscrivere il suo contenuto come 37, semplificando un po' il problema. Se la casella s[3][9] avesse in origine contenuto soltanto 135, allora, eliminando 1 e 5, il suo contenuto sarebbe stato forzato al valore 3.

Le doppie possono rivelarsi più insidiose: per esempio, si supponga che due caselle di una riga, colonna o griglia 3x3 abbiano come lista di valori possibili 2467 e 257 e che nessuna altra casella di quella riga, colonna o griglia 3x3 menzioni 2 e 7 come valori possibili. In questo caso 27 è una *doppia nascosta*: una delle due caselle deve assumere il valore 2 e l'altra il valore 7, quindi tutte le cifre diverse da 2 e 7 possono essere eliminate dalle liste delle possibilità per queste due caselle (ovvero, 2467 diventa 27 e 257 diventa 27, creando una coppia di doppie e semplificando così il problema).

Triple

Si consideri la colonna 5 della Figura H.7. Le caselle barrate potrebbero essere già fissate o potrebbero avere una lista di valori possibili. Si noti la presenza di *triple*: le tre caselle contenenti esattamente le stesse tre possibilità 467, ovvero, le caselle s[1][5], s[6][5] e s[9][5]. Se una di queste tre caselle alla fine assume il valore 4, allora le altre si riducono a doppie di 67; se una di queste tre caselle alla fine assume il valore 6, allora le altre si riducono a doppie di

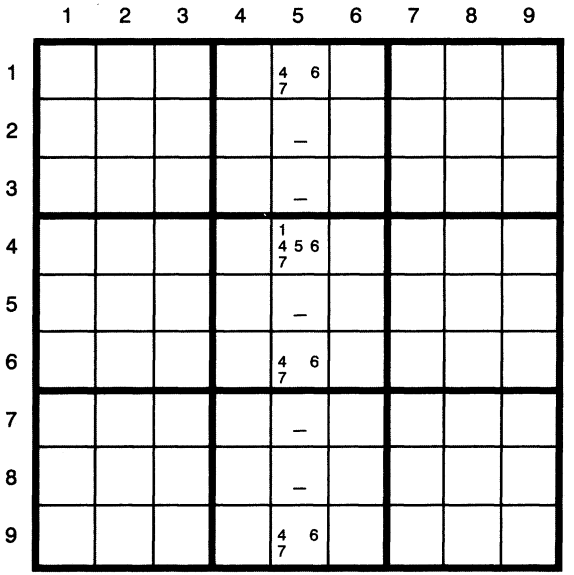


Figura H.7 Usare le triple per semplificare il puzzle

47; infine, se una di queste tre caselle alla fine assume il valore 7, allora le altre si riducono a doppie di 46. Fra le tre caselle contenenti 467, una alla fine deve assumere il valore 4, una il valore 6 ed una il valore 7. Quindi 4, 6 e 7 possono essere eliminati dalla casella s[4][5] che contiene le possibilità 14567, così possiamo riscrivere i suoi contenuti come 15, semplificando un po' il problema. Se la casella s[4][5] avesse in origine contenuto 1467, allora, eliminando 4, 6 e 7, il suo contenuto sarebbe stato forzato al valore 1.

Le triple possono rivelarsi più insidiose: per esempio, si supponga che delle caselle di una riga, colonna o griglia 3×3 abbiano come lista di valori possibili 467, 46 e 67. Ovviamente una di queste caselle deve assumere il valore 4, una il valore 6 ed una il valore 7. Quindi 4, 6 e 7 possono essere eliminati da tutte le altre liste di possibilità di quella riga, colonna o griglia 3×3.

Anche le triple possono essere nascoste: si supponga che una riga, colonna o griglia 3×3 contenga le liste di possibilità 5789, 259 e 13789 e che nessuna casella rimanente di quella riga, colonna o griglia 3×3 menzioni 5, 7 o 9. Allora una di quelle caselle deve assumere il valore 5, una il valore 7 ed una il valore 9. Definiamo 579 una *tripla nascosta* e tutte le possibilità diverse da 5, 7 e 9 possono essere eliminate da quelle tre caselle (ovvero, 5789 diventa 579, 259 diventa 59 e 13789 diventa 79), semplificando in qualche modo il problema.

Altre strategie di soluzione del Sudoku

Ci sono varie altre strategie di soluzione per il Sudoku. Di seguito elenchiamo due tra i numerosi siti, che possono aiutare ad analizzare più profondamente il problema, raccomandati nel Sudoku Resource Center (`www.deitel.com/sudoku`):

```
www.sudokuoftheday.com/pages/techniques-overview.php
www.angusj.com/sudoku/hints.php
```

H.4 Programmare dei risolutori di problemi Sudoku

In questa sezione vengono presentati dei suggerimenti su come programmare dei risolutori di Sudoku per mezzo di approcci diversi. Alcuni possono sembrare non intelligenti, ma, se riescono a risolvere dei Sudoku più velocemente di ogni persona sul pianeta, allora forse essi sono in qualche modo intelligenti.

Se avete risolto gli esercizi sul Giro del Cavallo (Esercizi 6.24, 6.25 e 6.29) e gli esercizi sulle Otto Regine (Esercizi 6.26 e 6.27), avete implementato vari approcci di risoluzione basati sulla forza bruta e su tecniche euristiche. Nelle prossime sezioni verranno suggerite delle strategie di risoluzione del Sudoku basate sulla forza bruta e su tecniche euristiche. Dovreste provare a programmarle come pure a crearne e ad implementarne di vostre. Il nostro obiettivo è semplicemente quello di rendervi familiari con il Sudoku ed alcune delle sue sfide e delle sue strategie di risoluzione. Procedendo con la lettura, diverrete più consapevoli nella manipolazione delle matrici e delle strutture di iterazione nidificate. Non abbiamo tentato di produrre delle strategie ottimali, così, una volta analizzate queste ultime, vorrete prendere in considerazione il problema di migliorarle.

Programmare una soluzione per Sudoku "facili"

Le strategie che abbiamo illustrato (eliminare delle possibilità in base a valori già fissati nella casella di una riga, colonna e griglia 3×3; semplificare un problema utilizzando i singoletti, le doppie, comprese quelle nascoste, le triple, comprese quelle nascoste) sono spesso sufficienti per risolvere un problema. Potete programmare le strategie ed iterarle finché tutte le 81 caselle saranno riempite. Per confermare che il puzzle compilato è un Sudoku valido, potete scrivere una funzione che controlli che ogni riga, colonna e griglia 3×3 contenga le cifre da 1 a 9 una ed una volta soltanto. Il vostro programma dovrebbe applicare le strategie in ordine; ognuna di esse o fissa il valore di una casella oppure semplifica un po' il problema. Ogni volta che una delle strategie funziona, ricominciate dall'inizio il vostro ciclo e riapplicate le strategie in ordine. Quando una strategia non funziona, provate la successiva. Per Sudoku "facili" queste tecniche dovrebbero generare una soluzione.

Programmare una soluzione per Sudoku più difficili

Nel caso dei Sudoku più complessi il vostro programma prima o poi raggiungerà un punto in cui vi saranno delle caselle non fissate con delle liste di possibilità; inoltre, nessuna delle strategie precedentemente discusse funzionerà. Se ciò accade, dapprima è opportuno salvare lo stato della scacchiera, poi si genera la prossima mossa scegliendo a caso uno dei possibili valori in una delle caselle rimanenti. Poi si rivaluta la scacchiera enumerando le possibilità rimanenti per ogni casella. A questo punto si possono applicare nuovamente le strategie di base, ciclando fra queste ripetutamente fintanto che il Sudoku viene risolto oppure si giunge nuovamente al punto in cui le strategie non migliorano ulteriormente la situazione sulla scacchiera; quindi si può nuovamente provare un'altra mossa a caso. Se si raggiunge il punto in cui ci sono ancora delle caselle vuote, ma non ci sono delle cifre ammissibili per almeno una di queste, il programma deve abbandonare quel tentativo, ripristinando lo stato della scacchiera al punto salvato in precedenza e ricominciando nuovamente l'approccio casuale. Il ciclo deve continuare fintanto che non viene trovata una soluzione.

H.5 Generare nuovi problemi di Sudoku

Inizialmente verranno considerati diversi approcci possibili per generare dei Sudoku 9×9 validi e completi con tutte le 81 caselle riempite. In seguito verrà suggerito come svuotare alcune caselle per creare dei problemi che i giocatori possono tentare di risolvere.

Approcci basati su "forza bruta"

Quando sul finire degli anni '70 comparvero i primi personal computer, essi erano in grado di elaborare decine di migliaia di istruzioni al secondo. I computer desktop al giorno d'oggi tipicamente elaborano miliardi di istruzioni per secondo ed il supercomputer più veloce del mondo può elaborare mille miliardi di istruzioni al secondo! Approcci del tipo forza bruta che avrebbero richiesto mesi di calcolo negli anni '70 oggi possono produrre delle soluzioni nel giro di qualche secondo! Questo fatto incoraggia chi vuole dei risultati velocemente a programmare dei semplici approcci del tipo forza bruta, ottenendo delle soluzioni più rapidamente di quanto richiesto per sviluppare delle più sofisticate strategie di risoluzione "intelligenti". Nonostante i nostri approcci a forza bruta possano sembrare lenti, essi sono in grado di produrre in modo meccanico delle soluzioni.

Per questo tipo di approcci ci sarà bisogno di qualche funzione accessoria. Si definisca la funzione

```
int validSudoku( int sudokuBoard[ 10 ][ 10 ] );
```

che riceve una scacchiera di Sudoku rappresentata come una matrice di interi (si ricordi che la riga 0 e la colonna 0 vanno ignorate). Tale funzione dovrebbe restituire 1 nel caso in cui una scacchiera completata sia valida, 2 nel caso in cui una scacchiera parzialmente completata sia valida e 0 altrimenti.

Un approccio di tipo forza bruta esaustivo

Un approccio di tipo forza bruta consiste semplicemente nel selezionare tutti i possibili assegnamenti delle cifre da 1 a 9 in ogni casella. Ciò potrebbe essere fatto con 81 comandi `for` annidati in cui ognuno cicla da 1 a 9. Il numero di possibilità (9^{81}) è talmente alto da indurre a pensare che non valga nemmeno la pena provare. Tuttavia tale approccio ha il vantaggio che prima o poi incapperà in ogni possibile soluzione, alcune delle quali potrebbero rivelarsi in modo fortuito abbastanza presto.

Una versione leggermente più intelligente di questo approccio a forza bruta esaustivo consiste nel verificare che ogni cifra che si sta per collocare lasci la scacchiera in uno stato valido. Se ciò accade, si procede piazzando una cifra nella prossima casella. Se la cifra che si sta per collocare lascia la scacchiera in uno stato non valido, allora si prova a collocare tutte le altre otto cifre in ordine su tale casella. Se una di esse funziona, ci si sposta alla prossima casella. Se nessuna di esse funziona, allora ci si sposta indietro alla casella precedente e si prova con il prossimo valore. Questo approccio può essere gestito in modo automatico mediante dei comandi `for` annidati.

Approccio di tipo forza bruta con permutazioni di righe selezionate casualmente

Ogni riga, colonna e griglia 3×3 in un Sudoku valido contiene una permutazione delle cifre da 1 a 9. Il numero di queste permutazioni è 9! (ovvero, 9·8·7·6·5·4·3·2·1 = 362.880). Si scriva una funzione

```
void permutations( int sudokuBoard[ 10 ][ 10 ] );
```

che riceva una matrice 10×10 e che nella porzione 9×9 di quest'ultima corrispondente ad una griglia di Sudoku riempia ognuna delle nove righe con una permutazione delle cifre da 1 a 9 determinata casualmente.

Il seguente è un metodo per generare casualmente delle permutazioni delle cifre da 1 a 9: per quanto riguarda la prima cifra, si scelga a caso una cifra da 1 a 9; per quanto riguarda la seconda cifra, si utilizzi un ciclo per generare ripetutamente a caso una cifra da 1 a 9 fintanto che non venga generata una cifra diversa dalla prima; per quanto riguarda la terza cifra, si utilizzi un ciclo per generare ripetutamente a caso una cifra da 1 a 9 fintanto che non venga generata una cifra diversa dalle prime due e così via.

Dopo aver sistemato nove permutazioni selezionate casualmente nelle nove righe del vettore di Sudoku, si richiami la funzione `validSudoku` sul vettore. Se restituirà 1, il compito sarà terminato. Se restituirà 0, bisognerà semplicemente generare in modo casuale altre nove permutazioni delle cifre da 1 a 9 da inserire nelle nove righe successive del vettore Sudoku. Questo semplice procedimento genererà dei Sudoku validi. A tal proposito, l'approccio garantisce che tutte le righe sono permutazioni valide delle cifre da 1 a 9; quindi bisognerebbe aggiungere un'opzione alla funzione `validSudoku` che controlli soltanto le colonne e le griglie 3×3.

Strategie risolutive euristiche

Quando è stato affrontato il Giro del Cavallo negli Esercizi 6.24, 6.25 e 6.29, è stata sviluppata un'euristica "dell'accessibilità": si ricordi che un'euristica è "una regola pratica". "Suona bene" e sembra una regola ragionevole da seguire. È programmabile, quindi fornisce un mezzo per pilotare un computer verso la risoluzione di un problema. Tuttavia gli approcci di tipo euristico non garantiscono necessariamente il successo. Per quanto riguarda problemi complessi come il Sudoku, il numero di assegnamenti possibili delle cifre da 1 a 9 è enorme, quindi la speranza riposta nell'utilizzo di un'euristica ragionevole consiste nel fatto che eviterà di sprecare tempo a considerare possibilità senza speranza, concentrandosi invece su tentativi di soluzione che in modo più probabile portino al successo.

Un'euristica basata sul "tenersi le possibilità aperte" per risolvere il Sudoku

Proviamo a sviluppare un'euristica basata sul "tenersi le possibilità aperte" per risolvere i Sudoku. In qualsiasi momento durante la risoluzione di un Sudoku, è possibile classificare la scacchiera elencando in ogni casella vuota le cifre da 1 a 9 che sono ancora delle possibilità aperte per tale casella. Per esempio, se una casella contiene 3578, allora deve prima o poi assumere il valore 3, 5, 7 oppure 8. Durante il tentativo di risolvere un Sudoku, si raggiunge un punto morto quando il numero delle cifre possibili che possono essere allocate in una casella vuota diviene pari a zero.

Quindi si consideri la seguente strategia:

1. Si associ ad ogni casella vuota una lista di possibilità costituita dalle cifre che possono ancora essere assegnate a quella casella.
2. Si caratterizzi lo stato della scacchiera contando semplicemente il numero dei possibili assegnamenti per l'intera scacchiera.
3. Per ogni possibile assegnamento relativo ad una casella vuota, si associ a tale assegnamento il conteggio caratterizzante lo stato della scacchiera dopo il piazzamento stesso.
4. In seguito si assegni la cifra alla particolare casella (fra tutte quelle che rimangono) che lasci il conteggio della scacchiera al più alto valore possibile (in caso di più assegnamenti con questa caratteristica, se ne scelga uno a caso). Questo è un modo per "tenersi le possibilità aperte".

Guristica di tipo lookahead ("guardare al futuro")

Questa strategia è semplicemente un abbellimento dell'euristica basata sul "tenersi le possibilità aperte". Nel caso di una decisione fra due assegnamenti alternativi, si guarda una mossa avanti. Si assegna quella particolare cifra nella particolare casella tale che gli assegnamenti successivi lasciano il conteggio della scacchiera con il valore più alto in seguito a due mosse.

Creare dei problemi di Sudoku con delle caselle vuote

Una volta che il proprio generatore di Sudoku è in esecuzione, si dovrebbe riuscire ad ottenere una gran quantità di Sudoku validi in breve tempo. Per creare un problema, si memorizza la griglia risolta ed in seguito si svuotano delle caselle. Un modo per fare ciò è quello di svuotare delle caselle in modo casuale. Un'osservazione generale è che i Sudoku tendono a diventare più difficili man mano che aumenta il numero di caselle vuote (anche se ci sono delle eccezioni a questo fenomeno).

Un approccio alternativo è quello di svuotare le caselle in modo che la scacchiera rimanga simmetrica. Ciò può essere ottenuto scegliendo in modo casuale una casella da svuotare e poi svuotando la "casella speculare". Per esempio, se si svuota la casella s[1][1] in alto a sinistra, si può svuotare anche la casella s[9][1] in basso a sinistra. Tali specularità sono calcolate preservando l'indice della colonna, ma determinando quello della riga sottraendo l'indice della riga iniziale da 10. Si possono determinare le specularità anche sottraendo contemporaneamente l'indice della riga e quello della colonna da 10. In questo caso la casella speculare a s[1][1] diventerebbe s[10-1][10-1], ovvero, s[9][9].

Una sfida di programmazione

I problemi di Sudoku che vengono pubblicati tipicamente hanno esattamente una soluzione, ma risulta piacevole risolvere qualsiasi Sudoku, anche quelli che hanno soluzioni multiple. Provate a sviluppare un metodo per dimostrare che un determinato problema di Sudoku ha esattamente una soluzione.

H.6 Conclusione

Questa appendice sulla soluzione e programmazione di problemi di Sudoku ha presentato diverse sfide. Accertatevi di consultare il Sudoku Resouce Center per le sue numerose risorse web che vi aiuteranno a padroneggiare il Sudoku ed a sviluppare vari approcci alla scrittura di programmi per creare nuovi problemi e per risolvere quelli esistenti.

Indice analitico

Simboli

— operatore, 75
!= operatore di disuguaglianza, 38
operatore del preprocessore, 23, 510
operatore del preprocessore, 510
#define direttiva del preprocessore, 190, 506
#define NDEBUG, 504
#error direttiva del preprocessore, 510
#ifdef direttiva del preprocessore, 509
#ifndef direttiva del preprocessore, 509
#include direttiva del preprocessore, 189, 505
#line direttiva del preprocessore, 511
#pragma direttiva del preprocessore, 510
#undef direttiva del preprocessore, 508
% operatore modulo, 34
%%, 338, 358
%c, 141, 200, 328, 366, 369
%d, 30, 32, 161, 353, 368
%E, 338, 351, 365, 368
%e, 354, 355, 368
%f, 68, 141, 354, 365, 368
%G, 354, 365, 368
%g, 354, 365, 368
%hd, 141
%i, 353, 368
%ld, 141, 163
%Lf, 141
%lf, 141
%lu, 141
%n, 357, 358, 368
%o, 353, 366
%p, 247, 357, 358, 367
%s, 357, 367, 369
%u, 141, 363, 366
%X, 353, 366
%x, 353, 366
& AND bitwise, 392, 394, 396
& operatore di indirizzo, 30
&= operatore di assegnamento bitwise AND, 450
(float), 66
(sequenza di escape del newline), 25
* carattere di soppressione dell'assegnamento, 372
* operatore di moltiplicazione, 34
*argv [], 519
*fgets, 520
++ operatore, 75, 76
+= assegnamento con addizione, 73, 74
-> operatore puntatore a struttura, 382
. operatore membro di struttura, 382
// commento su una singola linea, 538
< simbolo di redirezionamento dell'input, 516
<< scorrimento a sinistra, 392, 398
<= minore o uguale a, 38
<= minore uguale a, 38
<assert.h>, 143, 558
<ctrl> c, 527, 528
<errno.h>, 143, 557
<float.h>, 143, 591
<limits.h>, 143, 591
<locale.h>, 143, 559
<setjmp.h>, 143, 565
<signal.h>, 143, 527, 528, 534, 566
<stdarg.h>, 143, 518, 567
<stddef.h>, 143, 246, 265, 557
<time.h>, 143, 148, 151, 558
= operatore di assegnamento, 115
== operatore uguale a, 38, 115
> simbolo di redirezionamento dell'output ,516
>= maggiore o uguale a, 38
>> scorrimento a destra, 392, 398
>> simbolo di accodamento dell'output, 516
>>= operatore di assegnamento scorrimento a destra, 400
?, 167
'\0', 198
'\f', 308
'\n', 308
'\r', 308
'\t', 308
'\v', 308
^ operatore bitwise OR esclusivo, 392, 397
^= operatore di assegnamento bitwise OR esclusivo, 400
_ sequenza di escape del carattere apice singolo, 365
__DATE__ costante simbolica predefinita, 511
__FILE__ costante simbolica predefinita, 511
__LINE__ costante simbolica predefinita, 511
__STDC__ costante simbolica predefinita, 511
__TIME__ costante simbolica predefinita, 511
| OR inclusivo bitwise, 392, 397
| pipe, 516
|= operatore di assegnamento bitwise OR inclusivo, 400
||, 167
~ complemento bitwise a uno, 392, 397
0X, 364
0x, 364

A

a modo di apertura del file, 421
a+ modo di aggiornamento del file, 420
a+ modo di apertura del file, 386, 421
a.out, 15
ab modo di apertura di un file binario, 525
ab+ modo di apertura di un file binario, 525
abbreviazioni simili all'inglese, 6
abort, 511, 528
abs, 583
accatastamento dei comandi di controllo, 53
accesso non valido alla memoria, 528
accumulatore, 293
acos, 563
Ada, 7
add, 297
aggregati, 379
albero, 245, 379
albero binario, 449, 474
albero di ricerca binaria, 474, 479, 480, 487, 488
algebra, 34
algoritmo, 49, 55
algoritmo sorgi e splendi, 50
algoritmo completo, 65
allarme acustico (cicalino), 369
allarme(''), 25
allineamento, 352
allineamento a destra, 352
allineamento a sinistra, 352
allineare a destra gli interi, 359
allineare a sinistra le stringhe in un campo, 362
allineato a destra, 359
allocazione della memoria, 143
allocazione dinamica della memoria, 451, 530
altri argomenti, 352, 366
ambiente, 13
ambiente C, 13
ampersand (&), 31, 33
analisi dei dati di un'indagine, 210, 211, 212, 213, 214, 215
analisi del testo, 346
AND, 393
AND bitwise (&), 392, 393, 394, 395, 410
anno bisestile, 130
ANSI C, 2, 10, 19, 253, 511, 524
Appendice D, 104
Appendice E, I sistemi numerici, 308, 316
Apple Computer, 5
applicazioni commerciali, 7
applicazioni distribuite in ambiente client/server, 6
approcci brutali per il giro del cavallo, 240
approccio di costruzione a blocchi, 9
aprire un file, 415
area, 89
argc, 519
argomenti della riga di comando, 519
argomenti variabili, 513
argomento, 25, 133, 506
aritmetica dei puntatori, 267, 268
arrotondamento, 352
arrotondare verso lo zero, 551
arrotondare verso meno infinito, 551
arrotondato, 68
ASCII (Codice Standard Americano per lo Scambio delle Informazioni), 328, 415
asctime, 590
asin, 563
assegnare un nome, 92
assembler, 6
assert, 511
associano da destra a sinistra, 41, 68
associatività, 35, 41, 77, 114, 187, 401
asterischi iniziali, 348
asterisco (*), 34
astrazione, 134
atan, 563
atan2, 563
atexit, 522, 523, 582
atof, 313, 314, 578
atoi, 313, 314, 315, 578
atol, 313, 315, 579
attraversamento di un labirinto, 169, 301
auto, 154
auto-esplicativo, 29, 381
azione, 25, 37, 49, 58

B

B, 8
backslash (\), 365
backslash (\\), 25
base, 316
BCPL, 8
Bell Laboratories, 8, 10
binario, 128, 130, 308
bit, 413
blocco, 25, 54, 138
blocco di controllo dei file (FCB), 415, 417
blocco di dati, 335
blocco più esterno, 157
blocco più interno, 157
Böhm, C., 51
bombing, 64
Borland C++, 13, 522
branch, 494
branch negative, 494
branch zero, 494, 495, 498
break, 106, 107, 110, 111, 112, 130
bsearch, 583
bubble sort (ordinamento a bolle) 208, 209, 210, 231, 284
bucket sort, 242
Byron, Lord, 8
byte, 413

C

C, 1
C++, 2, 6, 10, 11, 12, 13, 139
C95, 537
C99, 535
calcoli, 4, 31, 41
calcoli finanziari, 101
calendario, 130
calloc, 530
campo, 414
campo di bit, 402, 403
campo di bit non denominato, 404
campo di bit non denominato con dimensione zero, 404
carattere, 413
carattere apice singolo ('), 357
carattere di escape, 25, 365
carattere di salto pagina ('), 308
carattere di soppressione dell'assegnamento (*), 372
carattere di sottolineatura (_), 29
carattere NULL di terminazione, 198, 199, 306, 308, 323, 324, 356, 357
carattere nullo, 198, 306
carattere punto interrogativo (?), 365
carattere virgolette (), 25
caratteri di controllo, 308
caratteri di delimitazione, 334
caratteri di scansione, 367
caratteri di spazio bianco, 53
caratteri letterali, 352
caratteri speciali, 306
caratteri stampabili, 308
caret (^), 370
caricamento, 13
caricare, 13, 500
caricare un programma in memoria, 293
caricatore, 13
case sensitive, 29
casi di base, 161
casinò, 145, 151
caso di default, 102, 104
cast, 503
Celsius, 377
char, 141, 306
char *, 357
chiamante, 132
chiamata di funzione, 132, 138, 139
chiamata e ritorno da funzione, 143
chiamata per riferimento 144, 203, 204, 248, 250, 252, 254, 261, 459
chiamata per riferimento simulata, 145, 203, 204, 248
chiamata per valore, 144, 248, 249, 250, 354
chiamata ricorsiva, 161
chiarezza, 18
chiarezza del programma, 1, 18
chiave del record, 414
chiave di ricerca, 216
chiusura anormale del programma, 528
cicalino, 25
ciclo, 63, 91
ciclo controllato da un contatore, 70
ciclo di esecuzione dell'istruzione, 297
ciclo infinito, 59, 67, 96, 109
cifra, 48
cifre, 543
cifre binarie, 413
cifre decimali, 413
circonferenza, 89
classe di memoria, 154
clearerr, 577
clock, 588
clock_t, 588
COBOL (COmmon Business Oriented Language), 7
coda, 243, 399, 449, 467, 468
Codice Americano Standard per lo Scambio delle Informazioni 104
codice dell'operazione, 293, 493
Codice di Scambio (EBCDIC), 328
codice in linguaggio macchina, 14
codice Morse, 349
codice Morse internazionale, 349
codice oggetto, 14, 15
codice ottimizzato, 501
codice portabile, 9
coercizione degli argomenti, 141
collegamento, 154, 155
collegamento esterno, 522
collegamento interno, 522
colonna, 222
comando, 489
comando cc, 15
combinazione di tasti per l'end-of-file, 516
comitato dello standard C, 139
Comitato Nazionale Americano per gli Standard dei Computer e l'Elaborazione della Informazione, 8
commento, 23
compilare, 13
compilatore, 14, 19, 23, 26, 24, 521
compilatore C, 23
compilatore GNU GCC, 536
compilatori ottimizzanti, 156
compilazione, 14
compilazione condizionale, 505, 508
complemento, 395, 396
complemento a uno, 550
complessità esponenziale, 168
_Complex, 548
complex, 548
componenti, 10
compromesso tempo/spazio, 258
computer, 3
Comunicazioni dell'ACM, 51
concatenazione di stringhe, 346

condividere le informazioni, 5
condizione, 38, 108
condizione di continuazione del ciclo 91, 92, 93, 94, 96, 108
condizione semplice, 112
condizioni di errore, 143
confine della unità di memoria, 404
confrontare le strutture, 381
confrontare le unioni, 390
const, 206, 253, 254, 260, 240
contare le lettere dei voti, 102
contatore, 60, 61
contatore del ciclo, 92
contatore di dati, 497
contatore di istruzioni, 497
conti di credito, 126
continue, 110, 111, 112, 130
contrassegnata, 494
controllo dei limiti, 195
controllo dei limiti di un vettore, 195
controllo del programma, 50
controllo di tipo, 140
conversione da binario a decimale, 548
conversione da decimale a binario, 549
conversione da decimale a esadecimale, 549
conversione da decimale a ottale, 549
conversione da esadecimale a binario, 547
conversione da esadecimale a decimale, 548
conversione da notazione infissa a polacca inversa, 483
conversione da ottale a decimale, 548
conversione da ottale in binario, 547
conversione esplicita, 68
conversione implicita, 68
convertire le lettere minuscole in maiuscole, 143
copia, 144
copia di stringhe, 269, 346
copia temporanea, 68
corpo, 25
corpo della funzione, 138
corpo di un while, 59
corpo di una funzione, 25
correggere, 18
coseno, 134
coseno trigonometrico, 134
cosh, 563
costante, 493
costante di carattere, 305, 357
costante di enumerazione, 151, 405, 508
costante di stringa, 306
costante simbolica, 105, 190, 505, 506, 512
costanti simboliche predefinite, 511
Costruire un compilatore, 492
costrutto, 509
CPU, 14, 15
craps, 138, 150, 151, 152, 153, 236
crashing, 64
creare frasi, 344
crivello di Eratostene, 242
ctime, 590
cursore, 365

D

data, 143
database, 414
dati, 3
DBMS, 414
debugger, 509
DEC PDP-11, 8
DEC PDP-7, 8
decimale, 128, 308, 316
decisione, 3, 4, 26, 37, 42, 59
decremento, 92
definizione di struttura, 379
definizione di una macro, 506
definizione iterativa, 161
definizione ricorsiva, 161
Deitel, H.M., ix, xvi
deriferimento, 245, 249
descrittore di file, 415
diagnostica, 504
diagnostiche, 143
diagramma di flusso, 51
diagramma di flusso di una struttura for, 97
diagramma di flusso elementare, 118, 119
diametro, 89
dichiarazione, 29, 30, 138, 497
dichiarazione di una unione, 389
differimento indefinito, 276, 291
difftime, 537
dimensione di campo, 101, 352, 359, 361, 371
dimensione di un campo di bit, 401
Dipartimento della Difesa (DOD), 16
dipendenza dalla macchina, 6, 404
direttive del preprocessore, 14, 506
diritti di accesso, 33
disco, 3, 4, 5, 14, 15
disegnare i diagrammi, 127
dispositivi, 3, 4, 14, 15
dispositivo, 4
dispositivo di input, 3
dispositivo di memoria secondaria, 3, 14
dispositivo di output, 4
dispositivo standard per l'errore (stderr), 16
dispositivo standard per l'input (stdin), 16
dispositivo standard per l'output (stdout), 16
distanza tra due punti, 182
div, 578
dividi e conquista, 131
divina proporzione, 165
divisione, 4, 35
divisione per zero, 64, 527, 528
divisione tra interi, 35, 67
dizionario, 448

documentare i programmi, 23
documento dello standard ANSI C, 19, 537
doppia indirezione, 459
doppio backslash (\\), 25
doppio uguale, 34
double, 141, 162, 524
dump, 298
dump del computer, 297

E

EBCDIC (Codice di Scambio Decimale Codificato in Binario Esteso) 328
eccezione di calcolo in virgola mobile 527
editor, 14, 15, 305
EDOM, 557
effetti collaterali, 144, 156, 168
efficienza, 10
elaborazione batch, 4
elaborazione del testo, 305
elaborazione di stringhe, 197, 198
elaborazione distribuita, 5
elaborazione interattiva, 31
elaborazione personale, 5
elemento della casualità, 145
elemento di un vettore, 185
elemento zero, 185
elenchi variabili di argomenti, 516, 518
elevamento a potenza, 37
elevamento di un intero a una potenza intera, 169
elevare al cubo una variabile, 251
eliminare un nodo da una lista, 461
eliminazione da una lista concatenata, 169
eliminazione dei duplicati, 242, 479, 487
eliminazione dei goto, 51
ellissi (...) in un prototipo di funzione, 516
emacs, 13
end-of-file, 105, 307, 319, 320
enumerazione, 151, 405, 406, 493
EOF, 105, 307
ERANGE, 557
errno, 557
errore del linker, 521,
errore di imprecisione di uno, 95, 186
errore di segmentazione, 33
errore di sintassi, 30, 58, 77, 113, 116, 137, 138
errore durante l'esecuzione, 198
errore fatale, 48, 64, 298
errore logico, 38, 88, 107, 115, 116, 139, 192, 389
errore logico fatale, 58
errore logico non fatale, 58
errore non fatale, 48, 139
errori a tempo di compilazione, 30
errori di compilazione, 30
errornum, 340
esadecimale, 128, 308, 317, 352, 358
esecuzione condizionale delle direttive del preprocessore 505
esecuzione sequenziale, 51
eseguire, 13, 14
esempio del libretto di risparmio, 99
esercizio del Pig Latin, 344
esercizio della limerick, 344
esercizio dell'indovina il numero, 199
espandere una macro, 506
espansione di una macro, 506
espressione, 96, 138, 506
espressione con i puntatori, 267
espressione condizionale, 55
espressione di controllo in uno switch, 105
espressione intera costante, 107
espressioni di tipo misto, 141
etichetta, 157, 531
etichetta della struttura, 380
etichette di case, 102, 105
Eulero, 238
euristica dell'accessibilità, 240
exit, 522, 529, 566
EXIT_FAILURE, 522, 582
EXIT_SUCCESS, 522, 582
extern, 156, 521

F

falso, 38
fare riferimento a un valore in modo diretto, 245
fare riferimento a un valore in modo indiretto, 245
fare riferimento a una variabile in modo indiretto, 245
fase di chiusura, 58, 65
fase di compilazione, 200, 206, 264
fase di elaborazione, 61, 65, 66
fase di esecuzione, 115, 139, 247
fase di inizializzazione, 61, 65, 66
fattore di scala, 145, 150
fattoriale, 70, 126, 162
fattoriale di n (n!), 162
FCB, 415, 419, 420
fclose, 570
feof, 418, 419, 428, 577
ferror, 577
fflush, 570
fgetc, 416, 448, 574
fgetpos, 576
FIFO (first-in-first-out), 467
figlio, 474
figlio destro, 474
figlio sinistro, 474
FILE, 558, 568
file, 413, 414
file ad accesso casuale, 427, 428, 433
file ad accesso sequenziale, 416
file aperto in modo binario, 525
file binario, 525
file delle transazioni, 445

file dello standard input, 415
file dello standard output, 415
file di intestazione, 24, 144, 506
file di intestazione <ctype.h>, 143, 307, 508, 504
file di intestazione <math.h>, 100, 133, 143, 509
file di intestazione <stdio.h>, 24, 105, 145, 159, 318, 351, 415, 433, 508
file di intestazione <stdlib.h>, 143, 313, 530, 525
file di intestazione <string.h>, 143, 323, 532
file di intestazione della libreria standard, 143, 505
file di intestazione matematico, 133
file di testo, 525
file header del C95, 537
file header del C99, 537
file offset, 423
file principale, 445
file sequenziale, 414, 423
file server, 5
file sorgente multipli, 520, 521, 522
file temporaneo, 525, 526, 527
fine di una coda, 449, 467
fine della immissione dei dati, 62
first-in-first-out (FIFO), 467
flag, 352, 362
flag #, 362, 364
flag +, 362, 363
flag -, 362
flag 0 (zero), 362, 364, 365
flag spazio, 362, 363
float, 67, 68, 141, 524
flusso di controllo, 41, 59
fopen, 418, 517
formato esponenziale, 352
formato tabulare, 189
FORTRAN FORmula TRANslator, 7
fprintf, 572
fputc, 416, 574
fputs, 448, 574
fratello, 474
fread, 428, 433, 576
frame dello stack, , 144
free, 451, 581
freopen, 571
frexp, 564
fscanf, 572
fseek, 430, 569
fsetpos, 571
ftell, 576
funzione, 9, 15, 24, 121, 131
funzione ceil, 134, 511
funzione chiamante, 132
funzione chiamata, 132
funzione cos, 134, 563
funzione definita dal programmatore, 132
funzione di Fibonacci, 169
funzione di libreria, 9
funzione esponenziale, 134
funzione exit, 522
funzione exp, 134, 564
funzione fabs, 134, 564
funzione fattoriale, 169
funzione fattoriale ricorsiva, 162
funzione floor, 134, 565
funzione fmod, 134, 565
funzione gcd ricorsiva, 181
funzione inline, 553
funzione iterativa, 218
funzione log, 134, 510
funzione log10, 134, 510
funzione non ricorsiva, 180
funzione pow (potenza), 37, 100, 134, 511
funzione predicato, 458
funzione ricorsiva, 161
funzione ricorsiva di elevamento a potenza, 180
funzione ricorsiva mazeTraverse, 302
funzione ricorsiva quicksort, 300
funzione sin, 563
funzione sqrt, 134, 564
funzione tan, 134, 563
funzioni della libreria matematica, 133, 143, 183
funzioni della libreria per l'input/output standard (stdio), 318
funzioni dello standard ANSI, 10
funzioni di confronto delle stringhe, 345
funzioni di confronto incluse nella libreria per la gestione delle stringhe, 326, 327
funzioni di ricerca incluse nella libreria per la gestione delle stringhe 328, 329
funzioni di utilità, 144
funzioni per caratteri e stringhe della libreria per l'input/output standard 319
funzioni per la conversione delle stringhe, 313
funzioni per la conversione delle stringhe incluse nella libreria di utilità generiche 313
funzioni per la manipolazione della memoria incluse nella libreria per la gestione delle stringhe, 323, 335, 336, 340
funzioni per l'elaborazione delle stringhe, 144
fwrite, 428, 429, 430, 576

G

generatore di cruciverba, 350
generatore di parole con i numeri telefonici, 447
generazione casuale di labirinti, 302
generazione di numeri casuali, 275, 344
gerarchia di dati, 414, 415
gerarchia di promozione, 141
gestione dei caratteri, 307, 558
gestione dei segnali, 527, 566
gestione delle stringhe, 323, 585
gestione dinamica della memoria, 245
gestore del segnale, 566
getc, 508, 574

getchar, 319, 416, 448, 508, 574
getenv, 582
gets, 319, 320, 574
giochi di carte, 291
gioco, 145
gioco dei dadi, 145, 151
giustificato a destra, 101
giustificato a sinistra, 101
giustificazione tipografica, 347
gmtime, 537
goto, 531, 532
grafico a barre, 127, 159
gruppo di scansione, 370
gruppo di scansione invertito, 370, 371

H

halt, 297
hardware, 2, 3

I

IBM, 5
identificatore, 29, 506
identificatore della macro, 506
if struttura di selezione, 38, 53
il giro del cavallo, 238
verifica del giro ciclico, 242
immagine, 15
immagine eseguibile, 15
incapsulamento delle informazioni, 157, 261
inclusione dei file di intestazione, 142
incrementare, 92
incrementare una variabile di controllo, 92
indicatore di end-of-file, 307
indicatore di EOF, 105
indicatori di conversione, 352
indicatori di conversione per caratteri e stringhe, 356
indicatori di conversione per gli interi, 353, 367
indicatori di conversione per i valori in virgola mobile, 355
indicatori di conversione per valori in virgola mobile, 354, 369
indice, 186
indipendente dalla macchina, 8
indipendente dall'hardware, 8
indirizzo, 459
indirizzo di un campo di bit, 405
indirizzo di una variabile, 33
ingegneria del software, 112, 157, 253
inizializzare i vettori multidimensionali, 222
inizializzare le strutture, 382
inizializzare un vettore, 189
inizializzare una unione, 390
inizializzatore, 189
inizializzatore designato, 543
inline, 552
input/output (I/O) , 515
inseminare la funzione rand, 148
inserimento dei caratteri letterali, 352
inserimento dello spazio, 48
inserimento in un albero binario, 169
inserimento in una lista concatenata, 169
inserire ed eliminare i nodi in una lista, 453
insieme di carattere, 328
insieme di caratteri, 413
insieme di caratteri ASCII, 104. 542
insigned long int, 141
installazione centrale di computer, 5
int, 141, 524
int implicito, 546
intercettare, 527
intercettare un segnale interattivo, 527
interesse composto, 99, 100, 101, 126, 128
intero, 28
intero decimale con segno, 353
intero decimale senza segno, 353
intero esadecimale senza segno, 353
intero ottale senza segno, 353
intero senza segno, 393, 524
intero unsigned long, 302, 524
interprete, 503
interrupt, 527
intestazione di funzione, 138
inventare nuovamente la ruota, 9, 132
inventario, 446
invocare una funzione, 132
ipotenusa di un triangolo rettangolo, 176
isalnum, 308, 309, 558
isalpha, 308, 309, 558
iscntrl, 308, 312, 313, 558
isdigit, 308, 309, 558
isgraph, 308, 312, 313, 558
islower, 308, 310, 558
isprint, 308, 312, 313, 558
ispunct, 308, 312, 313, 558
isspace, 308, 312, 313, 559
Istituto Nazionale Americano per gli Standard (ANSI), 2, 19, 594
istogramma, 127, 196
istruzione, 14, 25, 138, 449
istruzione add (aggiungere), 295, 459
istruzione assistita dal computer (CAI), 178
istruzione composta, 57
istruzione di assegnamento, 31
istruzione di azione, 51
istruzione goto, 51, 157, 531, 532
istruzione halt, 295, 494
istruzione illegale, 527, 528
istruzione load (caricare), 295
istruzione return, 138
istruzione store (immagazzina), 295
istruzione vuota, 58
istruzione write (stampa), 295

istruzioni di salto, 297
istruzioni di selezione, 129
istruzioni return senza espressione, 553
isupper, 308, 310, 558
isxdigit, 308, 309, 559
iterazione, 168, 169
iterazione controllata da un contatore, 60, 61, 91, 92, 93, 94
iterazione controllata da un valore sentinella, 62, 66, 67, 91
iterazione definita, 60, 91
iterazione indefinita, 63, 91
iterazioni determinate, 93

J

Jacopini, G., 51
job, 4

K

KIS (keepitsimple), 18

L

la tartaruga e la lepre, 292
labirinti di qualsiasi dimensione, 302
labs, 583
lancio dei dadi, 145, 147, 149
lancio di due dadi, 236
lancio di una monetina, 145, 179
last-in-first-out (LIFO), 461
ldexp, 564
ldiv, 578
le otto Regine, 170, 242, 244
approccio brutale, 242
le torri di Hanoi, 170, 181, 182
leggi di De Morgan, 129
leggibilità, 40, 71, 94, 136
lettera maiuscola, 29, 30
letterale, 20, 32
letterale composto, 543
lettere di sollecito, 350
lettere maiuscole, 48, 143
lettere minuscole, 48
lettura non distruttiva, 34
liberare la memoria, 451
libreria del C, xiii, 556
libreria di utilità generiche (stdlib), 314, 578
libreria per la gestione dei segnali, 527
libreria standard del C, 9, 13, 131, 144
librerie standard, 15
LIFO (last-in-first-out), 461
limiti, 591
limiti dell'implementazione, 539
limiti di credito, 126
limiti per la dimensione degli interi, 143
linea di flusso, 52
linee telefoniche, 5
linguaggi di alto livello, 6, 292
linguaggio assembly, 6
linguaggio C, 8
linguaggio di programmazione, 6
linguaggio Logo, 237
linguaggio macchina, 6, 14
Linguaggio Macchina del Simpletron (LMS), 293, 294, 302
linguaggio naturale di un computer, 6
linguaggio non tipizzato, 8
linguaggio portabile, 19
link, 15, 450, 452
linker, 14, 15, 26, 521
linking, 15
lista concatenata, 245, 379, 449, 452, 453
lista dei parametri, 136
lista di inizializzatori, 198
lista di inizializzatori di un vettore, 189
lista separate da virgole, 96
LMS, 294, 302
load, 297
localeconv, 560-561
località, 143
localizzazione, 559
localtime, 590
locazione, 33
logaritmo naturale, 134
long, 108, 163, 524
long double, 141, 524
long int, 141, 163, 524
long long int, 552
longjmp, 565-566
Lovelace, Lady Ada, 8
lvalue (valoredisinistra), 116, 186

M

macchina di Turing, 51
macchine desktop, 5
Macintosh, 418
macro, 142, 505, 506
macro definite in stdarg.h, 517
maggiore di due numeri, 82
main(), 24
main ricorsivo, 170, 182
make, 522
makefile, 522
malloc, 451, 530
manipolazione dei bit, 404
manipolazione del testo, 192
manipolazioni bitwise dei dati, 392
marcatore di end-of-file, 415
maschera, 393, 411
mascherare, 408
massimo comun divisore (MCD), 178, 181
matematica, 563

matrice, 222, 223
matrice immagazzinata, 415
matrice m per n, 222
mattoncini accatastati, 120
mattoncini nidificati, 120
maximum, 139, 140
mazzo di carte, 274
mblen, 583
mbstowcs, 584
mbtowc, 584
media, 46, 210
media aritmetica, 36
mediana, 210, 234
membro, 379
membro vettore flessibile, 552
memchr, 336, 339, 587
memcmp, 336, 338, 586
memcpy, 336, 585
memmove, 336, 337, 585
memoria, 3, 4, 13, 14, 33
memoria automatica, 155, 185
memoria primaria, 4, 14
memset, 339, 588
messaggi di errore, 16
messaggio, 25
metodo euristico, 240
mettere a punto, 51
mettere in coda, 463, 473
mktime, 589
Microsoft Visual Studio, 13
MingW, 536
moda, 214, 234
modello di azione/decisione, 26, 50
modello software, 296
modf, 564
modi di apertura, 421, 525
modifiche al compilatore di Semplice, 502
modifiche al simulatore Simpletron, 302
modo di apertura del file, 418
modo di apertura del file (w), 418
modulo, 43, 131
moltiplicazione, 4, 34, 35
moltiplicazione di due interi, 169
multipli di un intero, 89
multiprocessore, 13
multiprogrammazione, 5
multitasking, 8

N

n!, 161
NDEBUG, 512, 558
negazione logica, 112, 114
newline, 25, 26, 41, 53, 307, 308, 319, 320, 321, 342
nidificato, 71
nidificazione, 53, 69, 94, 118
nidificazione delle strutture di controllo, 53
nodi, 452
nodo foglia, 474
nodo radice, 474
nodo radice di un albero binario, 488, 489
nodo sostitutivo, 487
nome, 33, 92, 186
nome del file, 11
nome della funzione, 92, 116, 136, 156, 284
nome di etichetta di struttura, 381
nome di un vettore, 186
nome di una variabile, 92, 116, 245
nome di variabile, 29, 30, 77, 267
nome di variabile formato da più parole, 30
nomi dei parametri nei prototipi di funzione, 140
NOT logico, 112
notazione con complemento a due, 550
notazione con gli indici di vettore, 198, 293, 249
notazione con i puntatori, 273, 249
notazione con i vettori, 273
notazione con puntatore e indice, 270, 272
notazione con puntatore e offset, 270, 272
notazione esponenziale, 354, 355
notazione infissa, 483
notazione polacca inversa, 483
notazione posizionale, 544
notazione scientifica, 354
NULL, 188, 270, 306, 293, 319, 418, 450, 463, 508, 530, 557
numeri binari negativi, 550
numeri complessi, 547
numeri pseudocasuali, 148
numeri romani, 130
numero casuale, 143
numero di posizione, 185
numero di riga, 489, 493
numero esadecimale, 544
numero in virgola mobile, 62, 65, 69
numero ottale, 545
numero perfetto, 178
numero primo, 178

O

offset, 270, 432
oggetto, 10, 20
operando, 31, 294, 493
operatore !, 114
operatore &&, 112, 113, 168
operatore ||, 112, 113, 168
operatore binario, 31, 35
operatore bitwise OR (|), 395
operatore condizionale (?), 55, 77
operatore di assegnamento (=), 31
operatore di assegnamento addizione (+=), 75
operatore di complemento (~), 392
operatore di complemento bitwise (~), 392, 395, 396, 397

operatore di conversione, 68, 142
operatore di conversione (float), 66
operatore di decremento (—), 75
operatore di deriferimento (*), 144, 245, 247, 249
operatore di elevamento a potenza, 100
operatore di incremento (++), 75
operatore di indirizzo (&), 30, 144, 198, 246
operatore di negazione logica (!), 112, 115
operatore di postdecremento, 75
operatore di postincremento, 75
operatore di predecremento, 75
operatore di preincremento, 75
operatore di risoluzione del riferimento (*), 247, 383
operatore di scorrimento a destra (>>), 392, 410
operatore di scorrimento a sinistra (<<), 410
operatore di uguaglianza, 37
operatore freccia (->), 382
operatore logico AND (&&), 112, 113, 114, 394
operatore logico OR (||), 112, 113, 114, 397
operatore membro di struttura (.), 382, 383, 390
operatore modulo (%), 34, 35, 43
operatore puntatore a struttura (->), 382, 383, 390
operatore puntatore freccia (->), 382
operatore punto (.), 382
operatore sizeof, 264, 265, 266, 430, 448, 451, 508
operatore ternario, 55, 167
operatore unario, 68, 78, 264
operatore unario cast, 68
operatore unario sizeof, 264
operatore virgola (,), 96, 167
operatori, 74, 540
operatori aritmetici, 34
operatori aritmetici binari, 68
operatori bitwise, 392, 393
operatori bitwise di scorrimento, 398
operatori di assegnamento +=, -=, *=, /= e %=, 75
operatori di assegnamento bitwise, 400
operatori di uguaglianza, 37, 38
operatori logici, 112
operatori moltiplicativi, 68
operatori relazionali, 37, 38
operatori aritmetici di assegnamento, 75
operazioni aritmetiche, 294
operazioni di caricamento/immagazzinamento, 294
operazioni di input/output, 294
operazioni di trasferimento del controllo, 294
OR esclusivo bitwise (^), 392, 393, 394, 395, 396
OR inclusivo bitwise (|), 392, 393, 394, 395, 396
ora, 143, 588
ordinamento, 208
ordinamento a bolle utilizzando una chiamata per riferimento 261, 263
ordinamento dell'albero binario, 469
ordinamento per selezione, 169, 243
ordine, 49
ordine degli operandi degli operatori, 166, 167
ordine di chiusura da parte del sistema operativo, 527
Organizzazione Internazionale per gli Standard, 2
orologio, 148
ottale, 128, 130, 308, 316, 352
ottimizzare il compilatore di Semplice, 500
ottimizzato, 501
ovale, 52
overflow, 528
overflow dello stack, 144

P

p, 128
pacchetti di una rete di computer, 468
paga base, 6
paga lorda, 6
pagina logica, 365
palindromo, 243
parallelo, 8
parametro, 133, 136
parametro di tipo puntatore, 252
parentesi (), 30, 41, 187
parentesi graffa aperta ({), 24
parentesi graffa chiusa (}), 24, 26
parentesi graffe ({}), 57, 58
parentesi nidificate, 37
parentesi quadre ([]), 185, 186, 187
pari, 47
parole chiave, 41, 42, 59, 154
parti frazionarie, 67
Pascal, 2, 7
Pascal, Blaise, 7
passare i vettori alle funzioni, 202
passare un elemento di un vettore, 204, 205, 206
passare un vettore, 204, 205, 206
passo di ripartizione, 300
passo ricorsivo, 161, 300
PDP-7, 8
PDP-11, 8
permanenza, 154, 155, 157
permanenza automatica in memoria, 155, 200
permanenza in memoria, 154, 200
permanenza statica in memoria, 155
perror, 577
personal computer, 3
Personal Computer IBM, 5, 7
piattaforma hardware, 8
pila, 245, 379, 422, 423, 449, 461, 462
pipe (|), 516
piping, 516
Plauger, P. J., 9
poker, 291
poker a cinque carte, 291
polinomio, 37
pop, 142, 462
portabile, 8, 9, 19
portabilità, 8, 9, 16, 19
postincrementare, 75

potenza, 134
precisione, 68, 352, 360, 361
precisione di default, 68, 355
preincrementare, 75
prelevare, 297
prende, 38
prendere in input caratteri e stringhe, 369
preprocessore, 14, 142
preprocessore del C, 14, 24, 142, 190, 505
primo raffinamento, 63, 70
principio del minimo privilegio, 155, 207, 253, 263, 264
printf, 351, 572
priorità, 35, 41, 114, 187, 249
priorità degli operatori, 41, 249
priorità degli operatori aritmetici, 36
probabilità, 144
problema dei risultati dell'esame, 72
problema del calcolo della media, 61, 65, 66
problema del chilometraggio, 83
problema del limite di credito, 83
problema del linguaggio Semplice, 489
problema del numero maggiore, 46
problema del numero minore, 46
problema del numero telefonico, 345
problema del palindromo, 88
problema della codifica/decodifica, 90
problema della conversione da binario a decimale, 89
problema della matrice, 237
problema della paga straordinaria, 6, 85
problema della provvigione, 84
problema dell'else appeso, 87
problema dell'interesse semplice, 84
problema dell'interprete di Semplice, 503
procedura, 49
processo di compilazione, 497
prodotto, 42, 46
produzione, 500
profondità di un albero binario, 486
progettazione orientata agli oggetti (OOD), 13
programma copy su un sistema UNIX, 519
programma di editing, 13
programma di file-matching, 445
programma di mescolamento e distribuzione di carte, 275
programma di ordinamento, 281
programma di visualizzazione di un istogramma, 195
programma eseguibile, 26
programma oggetto, 21
programma per computer, 3
programma per il file-matching, 445
programma per il gioco dei dadi, 183
programma per il lancio dei dadi, 196
programma per il sondaggio degli studenti, 194
programma per la busta paga, 6
programma per la conversione metrica, 349
programma per la gestione del conto bancario, 436
programma per la gestione di una coda, 468
programma per la gestione di una pila, 462
programma per l'elaborazione delle transazioni, 435
programmatore, 13
programmatore di computer, 3
programmazione concorrente, 16
programmazione in linguaggio macchina, 293
programmazione modulare, 3
programmazione orientata agli oggetti (OOP), 13, 128
programmazione senza goto, 51
programmazione strutturata, 1, 3, 7, 12, 23, 41, 49, 51, 531
programmazione strutturata per raffinamenti successivi, 69
programmazione top down per raffinamenti successivi, 62, 63, 69, 275, 277
programmazione top-down per raffinamenti successivi, 3
programmi formati da diversi file sorgente, 154, 156
programmi orientati agli oggetti, 2
programmi strutturati, 12
promosso, 68
promozione, 68
prompt, 30
prompt della riga di comando di UNIX, 516
protezione degli assegni, 348
prototipo di funzione, 100, 136, 138, 139, 156
prototipo di funzione per printf, 516
provvigioni, 234
pseudocodice, 50, 72
puntatore, 245
puntatore a FILE, 420
puntatore a file, 415, 459
puntatore a puntatore, 420
puntatore a una funzione, 281, 282, 283
puntatore a void (void *), 269, 451
puntatore di posizione del file, 423, 433
puntatore generico, 269
puntatore NULL, 530, 531
punteggiatura, 243, 334
punto e virgola (;), 25, 34, 39
push, 142, 462, 466, 467
putc, 575
putchar, 319, 320, 321, 322, 327, 329, 330, 331, 332, 333, 334, 335, 337, 338, 339, 340, 341, 416, 508, 522
puts, 319, 320, 448, 522

Q

qsort, 583
qualificatore const, 253
qualificatore di tipo volatile, 524
quantità scalare, 204
quicksort, 170, 300

R

r modo di apertura del file, 421
r+ modo di aggiornamento del file, 420, 421
r+ modo di apertura del file, 421
radianti, 134
radice quadrata, 134
raggio, 89
raise, 527, 528, 566-567
rand, 144, 578
RAND_MAX, 144, 149
randomizzazione, 148
rapporto aureo, 165
rb modo di apertura di un file binario, 525
rb+ modo di apertura di un file binario, 525
read, 297, 493
realloc, 530, 581
record, 258, 416
record di attivazione, 144
redirezionamento, 351
redirezionare l'input da un file, 515
register, 154, 155, 156
registri hardware, 156
regola di accatastamento, 118
regola di nidificazione, 118
regole di promozione, 141
relazione gerarchica tra la funzione capo
    e quella operaia, 132
requisiti, 156
requisiti di efficienza, 156
restituire il controllo del programma, 132
resto, 134
rete di computer, 5
rete locale, 5
reti, 5
rettangolo, 52
return 0, 32
rewind, 523, 526
riassunto della programmazione strutturata, 116
ricerca, 216
ricerca binaria, 170, 180, 216, 217, 218, 219, 243
ricerca in un albero binario, 480
ricerca in una lista concatenata, 169
ricerca lineare, 170, 216, 217, 243
ricerca ricorsiva in una lista, 487
Richards, Martin, 8
richiesta di chiusura, 528
ricorsione, 160, 168, 169
ricorsione e iterazione, 168
ricorsione infinita, 163
riduzione in scala, 145
riempitivo, 404
rientrare, 94
rientro, 26, 53, 55, 56, 94
riferimenti irrisolti, 521
riga di comando di UNIX, 515
righe, 222
rimozione da un albero binario, 487
ripetizione controllata da un contatore, 94
risolvere il riferimento di un puntatore, 243, 250
risolvere il riferimento di un puntatore void *, 269
risparmiare la memoria, 401
Ritchie, D., 8, 19
ritorno carrello (', 308
riusabilità del software, 9, 26, 134, 138, 264, 521
rvalue (valore di destra), 116

S

salti non locali, 565
salto incondizionato, 499, 500
scacchi, 238
scacchiera, 48, 89
scalabile, 190
scalare, 204
scanf, 351, 573
schermo, 3, 4, 16
scorrimento a sinistra, 392
scostamento, 432
scrittura distruttiva, 33, 34
scrivere del codice, 13
scrivere in lettere l'importo di un assegno, 348
scrivere in un file, 417
secondo raffinamento, 62, 71
SEEK_CUR, 433, 569
SEEK_END, 433, 569
SEEK_SET, 433, 569
segnale, 527
segnale di attenzione interattivo, 528
segno di percentuale (%), 35
seme, 148
seno, 134
seno trigonometrico, 134
sequenza di escape, 25, 32, 365, 377
sequenza di escape del backspace, 365
sequenza di escape del carattere apice singolo, 365
sequenza di escape del carattere backslash, 25, 365
sequenza di escape del carattere punto interrogati, 365
sequenza di escape del carattere virgolette, 365
sequenza di escape del newline, 365
sequenza di escape del ritorno carrello, 365
sequenza di escape del salto pagina, 365
sequenza di escape della tabulazione
    orizzontale, 25, 365,
sequenza di escape della tabulazione verticale, 365
sequenza di escape dell'allarme, 25, 365
serie di Fibonacci, 165, 180
setbuf, 568
setjmp, 562
setlocale, 559
setvbuf, 568
Sezione speciale
    costruire il vostro compilatore, 449
    costruzione del vostro computer, 293

esercizi di manipolazione avanzata delle stringhe, 346
short, 108, 141
SIGABRT, 528, 566
SIGFPE, 528, 566
SIGILL, 566
SIGINT, 527-528, 566
signal, 512, 513, 528
signal_handler, 528
SIGSEGV, 528, 566
SIGTERM, 528, 566
simboli di connessione, 52
simboli speciali, 413
simbolo, 48
simbolo cerchietto, 52
simbolo di accodamento dell'output >>, 516
simbolo di azione, 52
simbolo di decisione, 52, 54
simbolo di redirezionamento dell'input <, 516
simbolo di redirezionamento dell'output >, 516,
simbolo ovale, 52
simbolo rettangolo, 52, 59, 106
simbolo rombo, 52, 54, 59, 106
Simpletron, 448
simulatore di computer, 296
simulazione, 144, 293
simulazione di un mescolatore e distributore di carte, 275, 386
simulazione di un mescolatore e distributore di carte ad alta efficienza, 386, 387, 388, 389
simulazione di un supermercato, 486
simulazione software, 296
sinh, 563
sinking sort (ordinamento con sprofondamento), 208
sistema di prenotazione per linee aeree, 237
sistema numerico binario, 543, 545
sistema numerico decimale, 543, 545
sistema numerico esadecimale, 543, 546
sistema numerico in base 2, 543
sistema numerico in base 8, 316
sistema numerico in base 10, 316, 543
sistema numerico in base 16, 316
sistema numerico ottale, 543, 545
sistema operativo, 4, 8, 13, 32
sistema per la gestione del database (DBMS), 414
sistemi guidati da menu, 285
sistemi numerici, 308
sistemi per la composizione tipografica, 305
sistemi per l'elaborazione di transazioni, 427
sistemi software di comando e di controllo, 8
size_t, 328, 265, 557
software, 2, 3
software commerciale, 7
software per l'impaginazione, 305
software riusabile, 10
sollecito, 350
somma, 4, 46
somma con for, 98, 99
somma degli elementi di un vettore, 169, 192
somma di due interi, 169
somma di numeri, 82
sommario della sintassi, 491
sondaggio, 193
sorgente, 20
sottoalbero destro, 474
sottoalbero sinistro, 474
sottrarre due puntatori, 269
sottrazione, 4
spaziatura verticale, 53
spazio, 41, 53, 308, 372
spazio bianco, 41, 308, 372
specifiche di classe di memoria, 154, 497
specifiche di conversione, 30, 32, 352
sprintf, 319, 322, 573
srand, 148, 580
sscanf, 319, 322, 573
stack 142
stack delle chiamate di funzione, 142
stack di esecuzione del programma, 142
stampante, 16
stampare, 16
standard error, 415
standard error (stderr), 352
standard input, 30, 318, 319, 515
standard output, 515
static, 155, 156, 157, 200
stddef.h, 143
stderr (il dispositivo dello standard error), 16, 415, 569
stdin (il dispositivo dello standard input), 15, 415, 569
stdlib, 313
stdout (il dispositivo dello standard output), 16, 415, 569
store (immagazzinare), 500
strcat, 323, 324, 586
strchr, 328, 329, 330, 587
strcmp, 326, 327, 586
strcoll, 560, 586
strcpy, 323, 324, 585
strcspn, 328, 330, 331, 587
stream (flusso), 351
stream dello standard input (stdin), 351
stream dello standard output (stdout), 351
strerror, 340, 578
strftime, 560, 590
stringa, 25
stringa di caratteri, 25, 187
stringa di controllo del formato, 30, 32, 352, 362, 366
stringa letterale, 199, 306
strlen, 340, 341, 588
strncat, 323, 324, 586
strncmp, 326, 327, 586
strncpy, 323, 324, 325, 585
Stroustrup, B., 10, 17
strpbrk, 329, 331, 332, 587

strrchr, 329, 332, 587
strspn, 329, 332, 333, 587
strstr, 329, 333, 334, 587
strtod, 313, 316, 579
strtok, 329, 334, 335, 587
strtol, 313, 317, 579
strtoul, 313, 318
struct, 185
struct_tm, 589
struttura, 258, 379
struttura di controllo, 51
struttura di controllo con un ingresso e una uscita singoli, 54, 117
struttura di controllo finché (while) 71
struttura di controllo se/altrimenti, 71
struttura di dati lineare, 474
struttura di iterazione, 51, 58, 59
struttura di iterazione do/while, 53, 108, 109, 110
struttura di iterazione finché, 71
struttura di iterazione for, 53
struttura di iterazione while, 58, 59
struttura di ripetizione do/while, 108
struttura di selezione, 51, 52
struttura di selezione doppia, 52, 56, 71
struttura di selezione if, 54
struttura di selezione if/else, 52, 54, 55
struttura di selezione multipla, 52, 106
struttura di selezione se/altrimenti, 71
struttura di selezione singola, 52, 54
struttura di selezione switch, 52
struttura di sequenza, 51, 52
struttura FILE, 420
struttura finché, 64, 71
struttura ricorsiva, 380, 450
struttura while, 53, 58
strutture dati last in, first out (LIFO) , 142
strutture di controllo nidificate, 69
strutture di dati, 449
strutture di dati dinamiche, 185, 245, 449
strutture if/else nidificate, 56
strutture statiche, 530
strxfrm, 560
su una riga, 35
supercomputer, 3
sviluppo del software, 3, 7
switch, 102, 106, 107
Symantec C++, 11
system, 582

T

tabella, 222
tabella dei file aperti, 415
tabella dei simboli, 492, 493
tabella di priorità, 543
tabella di verità, 113
tabulazione, 25, 26, 41, 48, 53, 365, 372
tabulazione verticale ('), 308
tangente, 134
tangente trigonometrica, 134
tanh, 564
tastiera, 3, 28, 30, 319
tasto di invio, 31
tasto enter, 31
tasto invio, 15, 107, 297
tasto return, 31
temperature Fahrenheit, 377
temporaneo, 100
terminale, 5
terminatore di istruzione (;), 25
terne pitagoriche, 128
testa di una coda, 449, 467
testa di una pila, 449 980, 506
Thompson, Ken, 8
time, 143, 148
time_t, 589
timesharing, 5
tipizzazione dei dati, 8
tipo, 33
tipo del valore di ritorno, 136
tipo del valore di ritorno void, 136
tipo di dato derivato, 379
tipo di struttura, 379
togliere dalla coda, 467, 473, 4734
token, 334, 493
token (simbolo), 491, 510
token in ordine inverso, 345
tolower, 308, 310, 311, 505
top, 63
totale, 60, 61
toupper, 308, 310, 311, 505
trasferimento del controllo, 51
traslare, 145
trovare il valore minimo in un vettore, 244
turtle graphics, 237
typedef, 385, 386

U

un altro problema dell'else appeso, 87
una stringa è un puntatore, 306
ungetc, 575
union, 389
unione, 389, 391, 401
unità aritmetica e logica (ALU), 4
unità di elaborazione, 3
unità di elaborazione centrale (CPU), 4
unità di input, 3
unità di memoria, 4
unità di memoria secondaria, 4
unità di output, 4
unità logiche, 3
UNIX, 6, 8, 13, 105, 418, 515, 519, 527
unsigned, 148
unsigned int, 141, 148, 149, 302

unsigned long int, 141, 580
utilizzo della memoria, 401

V

va_arg, 517, 567
va_end, 517, 567
va_list, 517, 567
va_start, 517, 567
valore, 31, 33, 186
valore assoluto, 134
valore chiave, 216
valore di segnalazione, 62
valore di traslazione, 150
valore di una variabile, 33
valore dummy, 62
valore finale di una variabile di controllo, 92, 95
valore flag, 62
valore immondizia, 60
valore iniziale di una variabile di controllo, 92
valore massimo, 85
valore minimo di un vettore, 169
valore posizionale, 544, 545
valore sentinella, 62, 67, 82
valore simbolo, 544
valutazione di un'espressione in notazione polacca inversa, 484
valutazione ricorsiva, 162
variabile, 29
variabile automatica, 155, 156
variabile di controllo, 91
variabile di struttura, 381
variabile di tipo puntatore, 267
variabile esterna, 156
variabile globale, 158, 159, 160, 521
variabile locale, 137, 144, 155, 156, 157, 158, 200
vera, 38
verificare se una stringa è palindroma, 169
versione standard del C, 8
vettore, 185, 186
vettore bidimensionale, 222
vettore di caratteri, 197, 198, 199
vettore di interi, 197
vettore di lunghezza variabile (VLA), 549
vettore di puntatori, 274
vettore di stringhe, 274
vettore dinamico, 530
vettore multidimensionale, 222, 223, 224
vettori di puntatori a funzioni, 302
vettori statici, 200
vfprintf, 573
vi, 13
violazione di accesso, 33, 307, 357
violazione di segmento, 527
virgola mobile, 354
virgolette, 32, 357
visibilità, 154, 157
visibilità nel blocco, 157, 158
visibilità nel file, 157
visibilità nel prototipo di funzione, 157, 158
visibilità nella funzione, 157
visita anticipata, 170, 475
visita con inOrder, 479
visita con postOrder, 479
visita con preOrder, 479
visita di un albero binario, 476
visita di una albero binario per livelli, 480
visita differita, 170, 475
visita in ordine (simmetrica), 169
visita per livelli, 488
visita per livelli di un albero binario, 488
visita simmetrica, 475
visualizzare, 16
visualizzare al contrario l'input della tastiera, 170
visualizzare al contrario un vettore, 170
visualizzare al contrario una lista concatenata, 170
visualizzare al contrario una stringa, 170
visualizzare al contrario una stringa immessa dalla tastiera, 170
visualizzare dei disegni, 126
visualizzare gli alberi, 489
visualizzare la ricorsione, 170, 181
visualizzare le date in vari formati, 347
visualizzare un albero binario, 489
visualizzare un quadrato, 88
visualizzare un quadrato vuoto, 88
visualizzare un vettore, 170, 244
visualizzare una stringa al contrario, 244
visualizzazione ricorsiva di una lista in ordine inverso, 487
void * (puntatore a void), 269, 335, 451
vprintf, 573

W

w modo di apertura del file, 421
w+ modo di aggiornamento del file, 420
w+ modo di apertura del file, 421
wb modo di apertura di un file binario, 525
wb+ modo di apertura di un file binario, 525
wcstombs, 585
wctomb, 584
Wirth, Nicklaus, 9
write, 297, 494

Z

zeri finali, 355